PhD-FSTC-2018-23
The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 06/04/2018 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

JULIAN THOMÉ
Born on 31 January 1986 in Prüm (Germany)

A SCALABLE AND ACCURATE HYBRID VULNERABILITY ANALYSIS FRAMEWORK

DISSERTATION DEFENCE COMMITTEE

DR. LIONEL BRIAND, Dissertation Supervisor
Professor, University of Luxembourg

DR. MEHRDAD SABETZADEH, Chairman
Senior Research Scientist, University of Luxembourg

DR. DOMENICO BIANCULLI, Vice Chairman
Research Scientist, University of Luxembourg

DR. GIOVANNI DENARO
Associate Professor, University of Milano-Bicocca, Italy

DR. ALESSANDRA GORLA
Assistant Research Professor, IMDEA Software Institute Madrid, Spain

In the loving memory of my dad

Abstract

As the Internet has become an integral part of our everyday life for activities such as e-mail, online-banking, shopping, entertainment, etc., vulnerabilities in Web software arguably have greater impact than vulnerabilities in other types of software. Vulnerabilities in Web applications may lead to serious issues such as disclosure of confidential data, integrity violation, denial of service, loss of commercial confidence/customer trust, and threats to the continuity of business operations. For companies these issues can result in significant financial losses.

The most common and serious threats for Web applications include *injection vulnerabilities*, where malicious input can be "injected" into the program to alter its intended behavior or the one of another system. These vulnerabilities can cause serious damage to a system and its users. For example, an attacker could compromise the systems underlying the application or gain access to a database containing sensitive information.

The goal of this thesis is to provide a scalable approach, based on symbolic execution and constraint solving, which aims to effectively find injection vulnerabilities in the server-side code of Java Web applications and which generates no or few false alarms, minimizes false negatives, overcomes the path explosion problem and enables the solving of complex constraints. In this work, we focus on Java because it is one of the most widely used technologies for Web development in the industrial context.

The main contributions of this thesis are:

- *Sound and scalable security auditing.* We define a specific security slicing approach for the auditing of security vulnerabilities in the server-side source code of Web applications which filters out irrelevant and secure code from the generated vulnerability report.
- *Search-driven constraint solving.* A search-driven technique for solving string constraints with complex string operations in the context of vulnerability detection.
- *Integrated Approach.* An integrated analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving.
- *Specialized Security Analysis.* The application of the above-mentioned techniques to detect the five most common types of injection vulnerabilities (Cross-Site-Scripting, SQL injection, XML injection, XPath injection, LDAP injection) in the context of Java applications.
- *Tool support.* The implementation of the proposed techniques in prototype tools.
- *Extensive empirical evaluation.* An extensive empirical evaluation of the approaches mentioned above.

Acknowledgements

The realization of this PhD project would have been unthinkable without the immense support by a number of great people to whom I would like to express my gratitude.

I would like to thank my supervisor Lionel Briand who made this PhD project possible in the first place. I am grateful for his invaluable support, feedback and guidance throughout the whole PhD project. It was a privilege to be part of the Software Verification and Validation lab.

Special thanks to my co-supervisors, Domenico Bianculli and Lwin Khin Shar. I am grateful for their encouragement, inspiration, and counsel in both personal and professional matters. Their devotion, help and advice were crucial for the realization of this PhD project. I am lucky to have had them as my mentors who taught me a lot about conducting rigorous research, writing papers and personal management.

I would like to express my gratitude to the members of my dissertation committee: Mehrdad Sabetzadeh, Giovanni Denaro, and Alessandra Gorla for their valuable time and remarks on my work. I would like to thank further Alessandra Gorla for keeping track of my progress and devoting her time to provide important feedback on my work.

I would also like to thank Wilhelm Meier (University of Applied Sciences Kaiserslautern, Germany), Andreas Zeller (Saarland University, Germany) and Alessandra Gorla who sparked my interest in research and encouraged me to pursue a PhD in the first place.

Furthermore, I would like to thank my family for believing in me. Special thanks to my beloved wife Lilit for her confidence in me, for her patience, love and support during the last years.

Last but not least, I would like to thank my colleagues and friends at the Software Verification Lab for the wonderful time we spent together. I am thankful for their helpful advice and for contributing to a positive work environment.

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03, and FNR9132112).

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ACG Attack Condition Generation.

ACO Ant Colony Optimization.

API Application Programming Interface.

CFG Control Flow Graph.

CWE Common Weakness Enumeration.

ESAPI Enterprise Security API.

EUF Equality of Uninterpreted Functions.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

ICFG Interprocedural Control Flow Graph.

IFC Information Flow Control.

JSON JavaScript Object Notation.

JSP Java Server Pages.

LDAP Lightweight Directory Access Protocol.

LDAPi LDAP injection.

LOC Lines of Code.

NVD National Vulnerability Database.

OWASP Open Web Application Security Project.

PDG Program Dependence Graph.

SAT Satisfiable.

SDG System Dependence Graph.

SQL Structured Query Language.

SQLi SQL injection.

SSA Static Single Assignment.

UNSAT Unsatisfiable.

URL Uniform Resource Locator.

XML Extensible Markup Language.

XMLi XML injection.

XPath XML Path Language.

XPathi XPath injection.

XQuery XML Query Language.

XSLT Extensible Stylesheet Language Transformations.

XSS Cross-Site-Scripting.

Part I

Overture

Chapter 1

Introduction

1.1 Motivation

As the Internet has become an integral part of our everyday life for activities such as e-mail, online-banking, shopping, entertainment, etc., vulnerabilities in Web software arguably have greater impact than vulnerabilities in other types of software. Vulnerabilities in Web applications may lead to serious issues such as disclosure of confidential data, integrity violation, denial of service, loss of commercial confidence/customer trust, and threats to the continuity of business operations. For companies these issues can result in significant financial losses [56].

This demonstrates that software security assurance is an important process in software development that protects the sensitive data and resources contained in and controlled by the software. Addressing security vulnerabilities early in the software development stage could decrease the cost of addressing them in later stages by a factor ranging between 30 and 60 times [84].

The most common and serious threats for Web applications include *injection vulnerabilities*, which are usually caused by improperly sanitized user inputs which are provided through *input sources* and used in security-sensitive program operations (*sinks*). These vulnerabilities can cause serious damage to a system and its users. For example, an attacker could compromise the systems underlying the application or gain access to a database containing sensitive information. According to the "Open Web Application Security Project (OWASP) Top 10 2017" vulnerability report [95], injection vulnerabilities are the most serious vulnerabilities for Web systems. Among them, Cross-Site-Scripting (XSS), SQL injection (SQLi), XML injection (XMLi), XPath injection (XPathi), and LDAP injection (LDAPi) vulnerabilities are the most commonly found in Web applications and Web services.

Symbolic execution and constraint solving represent a state-of-the-art approach used in security analysis to identify vulnerabilities in software systems. Symbolic execution executes a program with symbolic inputs and generates a set of *path condition*s. Each of them corresponds to a constraint imposed on the symbolic inputs to follow a certain program path. By solving

these constraints with a constraint solver, one can determine which concrete inputs can cause a certain program path to be executed.

In the context of security analysis for Web systems, this approach is used [64, 112, 40, 153] to detect injection vulnerabilities. Roughly speaking, this approach consists of solving *attack conditions*, i.e., the constraints obtained by conjoining the path conditions (generated by the symbolic execution) with attack specifications (what we are referring to as *threat models*) provided by security experts. The main strength of this approach is that vulnerability detection yields a limited number of false positives, since the concrete inputs determined with constraint solving prove the existence of vulnerabilities. However, the scalability, effectiveness and precision of this approach face two main challenges that affect symbolic execution and constraint solving [24]:

CH1 *(Path Explosion).* The path explosion problem is triggered by the huge number of feasible program paths that symbolic execution has to explore in large programs.

CH2 *(Solving Complex Constraints).* The problems related to solving complex constraints (e.g., constraints involving regular expressions or containing string/mixed or integer operations) are mainly due to the support for strings and their operations. In general, solving constraints that contain string operations requires to analyze the implementation of these operations, unless they can be treated as primitive functions in the constraint solver. However, there are typically thousands of string operations in a given programming language that cannot be solved because their semantics is not known to the constraint solver (e.g., `java.lang.String.regionMatches` and `java.lang.String.format`); we denote these operations as *unsupported operations*. Existing approaches support only a limited number of string operations (such as concatenation, assignment, and equality) as primitive functions. More complex operations have to be analyzed and transformed into an equivalent set of basic constraints containing primitive functions. This task is often not trivial and requires proficiency in the input language of the solver. A constraint solver that supports a limited set of operations can fail to solve constraints that contain unsupported operations, resulting in missed vulnerabilities.

Notice that while *CH1* and *CH2* are independent from the context in which symbolic execution and constraint solving are applied, the solutions to address them need to be tailored to a specific context.

1.2 Research Contributions

The ultimate goal of this work is *to provide a scalable approach, based on symbolic execution and constraint solving, which aims to effectively find injection vulnerabilities in the server-side code of Java Web applications and which generates no or few false alarms, minimizes false negatives, overcomes the path explosion problem and enables the solving of complex constraints.* In this work, we focus on Java because it is one of the most widely used technologies for Web development in the industrial context [26].

To achieve the above mentioned goal, the path explosion problem (*CH1*) and the challenge of solving complex constraints (*CH2*) have to be addressed first.

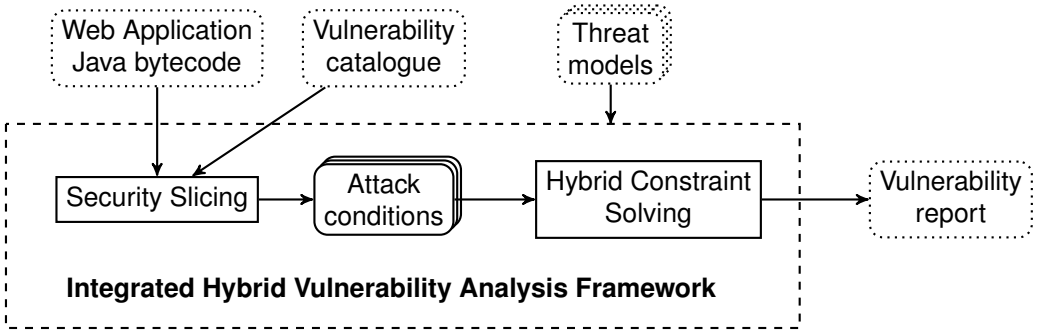


Figure 1.1: General overview of our approach.

For addressing *CH1* we propose *security slicing*, i.e., an approach to extract *security slices* from Java Web programs. A security slice contains a concise and minimal sequence of program statements that affect a given security-sensitive program location (sink), such as an SQL query statement. Given the bytecode of the Web application to analyze and the catalogue of vulnerabilities, it identifies sinks in the program and for each sink extracts only the security-relevant program parts leading to it. Since security slicing filters out program parts that are irrelevant to security, it can be used to reduce the manual effort in the context of security auditing, i.e., the examination of the source code for the purpose of detecting vulnerabilities. For the same reason it can be used to mitigate the path explosion problem in the context of symbolic execution. Symbolic analysis can be performed on security slices instead of the whole program; in this way path conditions are analyzed only with respect to the paths leading to sinks instead of every path in the program. Based on our experience, the number of sinks in a program is low[1] and security slices are much smaller (approx. 1%) than the program containing them.

For addressing *CH2*, we propose a *hybrid constraint solving* approach that orchestrates a constraint solving procedure for string/mixed and integer constraints with a search-based solving procedure which enables the constraint solver to solve *unsupported operations* and *complex string operations*.

Finally, we propose an *integrated hybrid vulnerability analysis framework* for injection vulnerabilities in Web applications, which leverages the synergistic combination of the two aforementioned steps: security slicing and hybrid constraint solving.

Our integrated approach is outlined in Figure 1.1 where dotted rounded rectangles correspond to global inputs/outputs, solid rounded rectangles correspond to intermediate inputs/outputs, solid rectangles correspond to operations, and the dashed rectangle correspond to a macro-step. It shows the "Integrated Hybrid Vulnerability Analysis Framework" macro-step that is composed of the "Security Slicing" and "Hybrid Constraint Solving" steps.

The first step ("Security Slicing") performs security slicing, i.e., given the bytecode of the Web application to analyze and the catalogue of vulnerabilities, it identifies sinks in the program and for each sink computes the path condition leading to it. This information is then used, together with the list of threat models, to generate attack conditions, i.e., conditions that could trigger a security attack over a security slice.

[1]Our experiments show that, on average, there are only 3 sinks in a Web program, related to the type of vulnerabilities we consider.

The second step ("Hybrid Constraint Solving") takes as input the attack conditions generated in the previous step, in the form of a constraint. The resulting constraint is then given as input to a *hybrid constraint solver*. The results yielded by the hybrid constraint solver are used to create the vulnerability report.

To summarize, the specific contributions of this thesis are:

I *Sound and scalable security auditing.* We define a specific security slicing approach for the auditing of security vulnerabilities in the server-side source code of Web applications. Like taint analysis, our approach also uses static program analysis techniques, which are known to be scalable [137]. However, our analysis additionally extracts control-dependency information, which is often important for the security auditing of input validation and sanitization procedures. Additionally, it filters out irrelevant and secure code from the generated vulnerability report. This ensures soundness and scalability.

II *Search-driven constraint solving.* A search-driven technique for solving string constraints with complex string operations in the context of vulnerability detection.

III *Integrated Approach.* An integrated analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving. This technique is general and language-independent.

IV *Specialized Security Analysis.* The application of the above-mentioned techniques to detect the five most common types of injection vulnerabilities (XSS, SQLi, XMLi, XPathi, LDAPi) in the context of Java Web applications.

V *Tool support.* The implementation of the proposed techniques in prototype tools: *JoanAudit*, i.e., the tool which implements our security slicing approach, *ACO-Solver* and *JOACO-CS*, i.e., the tools that implement hybrid constraint solving, and *JOACO*, i.e., the implementation of the integrated approach for vulnerability detection.

VI *Extensive empirical evaluation.* An extensive empirical evaluation of the approaches mentioned above.

1.3 Dissemination

Our research work has led to the following publications (listed in chronological order based on their publication date):

P1 THOMÉ, J. A scalable and accurate hybrid vulnerability analysis framework. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops* (2015), IEEE, pp. 61–62

P2 THOMÉ, J., SHAR, L., AND BRIAND, L. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities. In *Proceedings of the International Symposium on Software Reliability Engineering* (2015), IEEE, pp. 553–564

P3 THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. Search-driven string constraint solving for vulnerability detection. In *Proceedings of the International Conference on Software Engineering* (2017), IEEE, pp. 198–208

P4 THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of the Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 1004–1008

P5 THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software 137* (March 2018), 766–783

P6 THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. An integrated approach for effective injection vulnerability analysis of Web applications through security slicing and hybrid constraint solving. Tech. Rep. TR-SNT-2017-4, SnT Centre, 2018 ***(under submission)***

P1 provides a roadmap for the whole project; P2, P4 and P5 present our security slicing approach and are the basis for Chapter 3; P3 presents our search-driven constraint solving approach and is the basis for Chapter 4; the unpublished report P6 describes our integrated approach for injection vulnerability analysis of Web applications and is the basis for Chapter 5.

1.4 Organisation of the Thesis

In the remainder of Part I, i.e., in Chapter 2, we introduce some background concepts which are used throughout this thesis.

Part II is organized according to the steps illustrated in Figure 1.1: Chapter 3 explains our security slicing approach in detail; Chapter 4 introduces hybrid constraint solving; Chapter 5 illustrates the synergistic combination of security slicing, symbolic execution and constraint solving; Chapter 6 discusses the related work.

In the final part of the thesis (Part III), we provide conclusions and directions for future work in Chapter 7.

Chapter 2

Background

This chapter presents several background concepts which are used throughout this thesis; it is organized as follows: Section 2.1 introduces security concepts; Section 2.2 introduces concepts related to static analysis and program slicing; Section 2.3 provides definitions for constraints and constraint networks, and Section 2.4 provides insights into the Ant Colony Optimization (ACO) meta-heuristic.

2.1 Security

This section illustrates various security concepts and is structured as follows: Section 2.1.1 introduces injection vulnerabilities such as XSS, SQLi, XMLi, XPathi and LDAPi vulnerabilities; Section 2.1.2 explains the concepts of sources and sinks; Section 2.1.3 defines threat models.

2.1.1 Injection vulnerabilities

Injection vulnerabilities are program locations in which certain malicious input can be "injected" into the program to alter its intended behavior or the one of another system. An injection may occur when the user input is passed through the program to an interpreter or to an external program (e.g., a shell interpreter, a database engine) and the input data contain malicious commands or command modifiers (e.g., a shell script, an additional constraint of an SQL query). An injection vulnerability arises when the input is not properly validated or sanitized in correspondence of a sink.

Injection vulnerabilities can cause serious damage to a system and its users. For example, an attacker could compromise the systems underlying the application or gain access to a database containing sensitive information. The "OWASP Top 10 2017" report [95] shows that injection vulnerabilities represent the most common application security risk for Web applications.

There are several types of injection vulnerabilities. In this work we focus on the following five types, for which we give a short overview and an example based on the Common Weakness Enumeration (CWE) [32].

2.1.1.1 Cross-Site-Scripting (XSS)

It is an attack technique that injects malicious scripts into a trusted Web application. It can be accomplished by inserting untrusted, browser-executable data (e.g., JavaScript code, HTML tags) through a web request. When these data are used to dynamically generate a page requested by an end-user, the malicious script (injected through the untrusted data) will be executed in the user's browser, which is misled to consider the script as coming from a trusted source (and thus safe to execute). This untrusted code, once executed in the browser, may access and transmit to the attacker confidential information such as the user's session cookies (possibly leading to the hijacking of the user' session), or may alter the presentation of the content (possibly leading to phishing attacks).

As an example, consider the code snippet below:

```
1 String name = req.getParameter("user");
2 res.println("<div>Welcome␣"+name+"!</div>");
```

It dynamically generates an HTML element `div` based on the input received (through a Web request) and stored in the `name` variable. An attacker could perform an XSS attack by providing as input the string `<script language="JavaScript">alert('XSS');</script>`, which contains a snippet of JavaScript code. This code will be executed by the browser when it interprets the HTML code provided in the HTTP response. While in this case the injected code just displays a pop-up dialogue, in principle it could have much more harmful effects, like the ones mentioned above.

2.1.1.2 SQL injection (SQLi)

It is an attack technique that injects an SQL query in the input of a program, in order to read-/write/admin a relational database by affecting the execution of predefined SQL statements. It can be accomplished by placing a meta-character into the input string, which acts as a modifier of the original SQL statement and allows the attacker to alter its behavior.

As an example, consider the code snippet below:

```
1 String userid = req.getParameter("userid");
2 String query = "SELECT␣*␣FROM␣users␣WHERE␣user='"
3     + userid + "'";
4 Statement st = conn.createStatement();
5 ResultSet rs = st.executeQuery(query);
```

It dynamically builds a query string based on the input received (through a Web request) on the first line, by concatenating a constant string with the user input string. If a malicious user provides as input the string `name' OR 1=1 -`, the resulting query string will be `SELECT * FROM users WHERE user='name' OR 1=1 -'`. Notice that the malicious input string is built to correctly enclose (with a single quote character) the first condition of the `WHERE` clause and to add a second condition `OR 1=1`. The latter represents a tautology and causes the `WHERE` clause to always evaluate to true. The query becomes logically equivalent to `SELECT * FROM users`, allowing the attacker to access all the contents of the table `users` in the database.

2.1.1.3 XML injection (XMLi)

It is a technique that allows attackers to change the structure or the contents of an XML document before it is processed by the program. It can be accomplished by placing reserved words or meta-characters into the input string. Such an attack may yield various consequences, such as invalidating the XML document, injecting malicious content in the document, or forcing the XML parser to access external entities.

Consider, for example, the following XML document named `students.xml`:

```
<students>
 <student>
  <sid>1</sid>
  <email>wd@svv.lu</email>
  <uid>wd003</uid>
  <pwd>300wd</pwd>
 </student>
 <student>
  <sid>2</sid>
  <email>abf@svv.lu</email>
  <uid>abf004</uid>
  <pwd>400abf</pwd>
 </student>
</students>
```

and the following Java snippet that updates the email address of a student:

```
File db = new File("students.xml");
Document doc = DocumentBuilderFactory.newInstnce()
                .newDocumentBuilder().parse(db);
String uid = req.getParameter("uid");
String pwd = req.getParameter("pwd");
String emailnew = req.getParameter("emailnew");
//code to find the right <student> element and
//its children
if (student-uid.equals(uid) &&
    student-pwd.equals(pwd)) {
  if ("email".equals(node.getNodeName())) {
     node.setTextContent(emailnew);
  }
}
```

A malicious user could invalidate the XML document by entering an email address that contains a meta-character, such as an angular parenthesis like <. For example, if the attacker enters the email `wd@svv.lu<`, the corresponding element updated by the snippet above will look like `<email>wd@svv.lu<</email>` and will invalidate the document, possibly leading to data integrity issues.

2.1.1.4 XPath injection (XPathi)

It is an attack technique that injects an XPath query in the input of a program, in order to query or navigate an XML document. This attack can be accomplished by placing a meta-character into the input string, which alters the behavior of the original query by modifying the query logic or bypassing authentication. XPathi can be exploited directly by an application to query

an XML document as part of a larger operation, such as applying an XSLT transformation or an XQuery to an XML document.

As an example, consider the aforementioned document `students.xml` and the snippet of Java code below, which retrieves the student identification number with an XPath query:

```
1 File db = new File("students.xml");
2 Document doc = DocumentBuilderFactory.newInstance()
3                .newDocumentBuilder().parse(db);
4 XPath xpath = XPATHFactory.newInstance().newXPath();
5 String query = "//students/student[uid/text()='"
6                        + req.getParameter("uid")
7                        + "' and pwd/text() = '"
8                        + req.getParameter("pwd")
9                        + "']/sid");
10 NodeList nl = (NodeList) xpath.evaluate(query,doc);
```

The XPath query is built dynamically using the inputs received through a Web request. An attacker could access all the student identification numbers by simply entering as user id the string `foo' or 1=1 or 'a'='a` and any random password. In this way, the XPath query will look like

```
//students/student[uid.text()='foo' or 1=1
  or 'a'='a' and pwd.text()='nopwd']/sid
```

and the conditions of the selection will always evaluate to true, returning all the nodes and thus possibly leaking confidential information.

2.1.1.5 LDAP injection (LDAPi)

It is an attack technique that targets programs that build LDAP statements based on user input. The attack can be accomplished by inserting meta-characters or crafted LDAP filters that alter the logic of the query. As a consequence, permissions can be granted for unauthorized queries or for modifying the LDAP tree.

As an example, consider the code snippet below, extracted from an LDAP-based authentication system:

```
1 DirContext ctx = new InitialDirContext(env);
2 String userid = req.getParameter("userid");
3 String pwd = req.getParameter("pwd");
4 String base = "OU=snt,DC=uni,DC=lu";
5 String filter = "(&(sn=" + userid + ")(password="
6   + pwd + "))";
7 SearchControls ctls = new SearchControls();
8 NamingEnumeration<SearchResult> results =
9   ctx.search(base, filter, ctls);
```

where `env` is a `HashTable` object containing the environment properties for the LDAP connection. The `filter` object is dynamically constructed using the user input strings (`userid` and `pwd`) and then used for querying the LDAP server. If an attacker knows a valid user id (e.g., `"brianli"`), he can make an attack by entering a user id of the form `briandli)(&)`, and any value for the password (e.g., `"nopwd"`). This malicious string makes the filter string look like

`(&(sn=briandli)(&))(password=nopwd))`. Since an LDAP server processes only the first filter, the query will return true and will grant access to the attacker, even if he does not know the password of user `briandli`.

2.1.2 Input Sources and Sinks

Input sources are operations that access external data that can be manipulated by malicious users. Specifically, in our approach, we define as input sources the accesses to: HTTP request parameters (e.g., `getParameter`), HTTP headers, cookies, session objects, external files, and databases.

Sinks are operations that are sensitive to XSS, SQLi, XMLi, XPathi, or LDAPi. Specifically, we define the following elements as sinks:

- HTML document operations (e.g., `javax.servlet.jsp.JspWriter.print`);
- SQLi queries (e.g., `java.sql.Statement.executeQuery`);
- XML document operations (e.g., `org.xml.sax.XMLReader.parse`);
- XPath queries (e.g., `javax.xml.xpath.XPATH.evaluate`);
- LDAPi queries (e.g., `com.novell.ldap.LDAPConnection.search`).

We now illustrate XMLi and XPathi vulnerabilities and the concepts of input sources and sinks using the example in Listing 2.1.

```
1 protected void doPost(HttpServletRequest req, ...) {
2   String account = req.getParameter("account");
3   String password = req.getParameter("password");
4   String mode = req.getParameter("mode");
5   if(mode.equals("login")) {
6     allowUser(log,account, password);
7   } else {
8     createUser(log,account,password);
9   }
10 }
11 protected boolean allowUser(String account,
12   String password)  {
13   Document doc = builder.parse("users.xml");
14   XPath xpath = xPathfactory.newXPath();
15   String q = "/users/user[@nick='"+
16     ESAPI.encoder().encodeForXPath(account) +
17     "' and @password='" + ESAPI.encoder().encodeForXPath(password) + "']";
18   NodeList nl = (NodeList)xpath.evaluate(q, doc, XPathConstants.NODESET);
19 }
20 protected void createUser(String account,
21   String password) {
22   String newUser = "<user nick=\"" + ESAPI.encoder().encodeForXMLAttribute(account) +
23     "\" password=\"" + ESAPI.encoder().encodeForXMLAttribute(password) + "\" />";
24   FileWriter fw = new FileWriter("users.xml");
25   String newXML = "<users>\n" + getPresentUsers() + newUser + "\n</users>";
26   fw.write(newXML);
27 }
```

Listing 2.1: Secure servlet with sanitization functions.

The Java code snippet illustrated in Listing 2.1 grants or denies access to a Web application or service and/or creates a new user. The Java servlet interface implementation `doPost` stores the values of three POST parameters (`account`, `password`, and `mode`) in variables that carry the same names. All the parameters are provided by the user of the Web application. If the `mode` parameter is equal to the string `login`, function `allowUser` is called with `account` and `password` as parameters, to allow the user to access the application; otherwise, a new user account is created by invoking function `createUser` with `account` and `password` as parameters. We assume that users credentials are stored in the XML document shown in Listing 2.2 and named `users.xml`.

```
1 <users>
2         <user nick="alice" password="alicepass"/>
3         <user nick="bob" password="bobpass"/>
4 </users>
```

Listing 2.2: The user file `users.xml`

The accesses to HTTP parameters at lines Lines 2–4 are input sources. The XPath query at Line 18 and the XML document processing operation at Line 26 are sinks.

For granting or denying access, function `allowUser` in Listing 2.1 executes the XPath query (sink) at Line 18. This query compares the password, which is stored in the XML attribute `password`, for one of the entries in `users.xml` with the one accessed from an input source (the POST parameter `password`). In the example, the user inputs are sanitized at Lines 16–17 by invoking methods from the OWASP Enterprise Security API (ESAPI) [94], which provides a rich set of sanitization functions for various vulnerability types. If the user input was used directly in the sink without such sanitization, the sink could be subject to XPathi attacks. For example, in the case of `users.xml`, by just knowing a user name, an attacker could launch a tautology attack using the value `' or '1' = '1` as `password`, gaining access to the user's credential data.

Likewise, in the absence of any sanitization, the operation at Line 26 would be vulnerable to XMLi attacks. More specifically, at Line 26 an XML tag is created with a user input using string concatenation. If the user inputs stored in `account` and `password` were not sanitized, as they are at lines Lines 22–23, a user could compromise the integrity of the XML file by using one of the following meta-characters: `< > / ' = "`.

2.1.3 Threat Models

A *threat model* describes possible attacks that can be conducted through an input used in a sink. If an input can potentially contain values that match a threat model, the sink that uses such an input should be marked as vulnerable. According to our definition of a threat model and based on the types of vulnerabilities we focus on in this thesis, an attacker is not required to know the source code of the application; we only assume that the attacker is capable of providing input to the potentially vulnerable Web application through an input source.

In our approach we support the threat models listed in Table 2.1, which are based on various attack patterns defined as part of the OWASP security project [95]; for each model we indicate the corresponding constraint on the input. They are grouped by the type of sink (i.e., the type of vulnerability they exploit), and for each sink type we indicate also various *contexts*, which denote the possible ways in which inputs can be used in a sink. Each threat model is

Table 2.1: Constraints corresponding to threat models.

No.	Sink type	Context	Constraint
1	XSS	Element content: <tag>*input*</tag>	*input*.matches(".*[<>/].*")
2		Event handler value: <... onclick="*input*">	*input*.matches(".+")
3		Source value: <iframe src="*input*">	*input*.matches(".+")
4		Attribute value with single quotes:	*input*.matches(".*'.*")
5		Attribute value with double quotes:	*input*.matches(".*\".*")
6		Attribute value without quotes:	*input*.matches(".*[=<>/,;+-%*\[\]].*")
7		URL parameter value: <a href="http://...?param=*input*">	*input*.matches(".*['\"=<>/,;+-&*\[\]].*")
8	SQLi	Attribute value with single quotes: SELECT column From table WHERE row='*input*'	TAUTCSTR(*input*, "'")
9		Attribute value with double quotes: SELECT column From table WHERE row="*input*"	TAUTCSTR(*input*, "\"")
10		Attribute value with date delimiters: SELECT column From table WHERE row=#*input*#	TAUTCSTR(*input*, "#")
11		Attribute value without quotes or delimiters: SELECT column From table WHERE row=*input*	TAUTCSTR(*input*, "")
12	XMLi	Element content: <node>*input*</node>	*input*.matches(".*[<>].*")
13		CDATA content: <![CDATA[*input*]]>	*input*.matches(".*\]\]>.*")
14		Attribute value with single quotes: <node attr='*input*'/>	*input*.matches(".*'.*")
15		Attribute value with double quotes: <node attr="*input*"/>	*input*.matches(".*\".*")
16		Attribute value without quotes: <node attr=*input*/>	*input*.matches(".*['\"<>].*")
17		External entity: <!ENTITY xxe SYSTEM "*input*">]>	*input*.matches(".+")
18	XPathi	Attribute value with single quotes: //table[column='*input*']	TAUTCSTR(*input*, "'")
19		Attribute value with double quotes: //table[column="*input*"]	TAUTCSTR(*input*, "\"")
20		Attribute value without quotes or delimiters: //table[column=*input*]	TAUTCSTR(*input*, "")
21	LDAPi	LDAP search: search="(attr=*input*)"	*input*.matches(".*[()|*&].*")

indicated for a specific context of a specific sink type, to reflect the specialization of an attack to exploit a certain vulnerability with a particular input. Notice that a precise characterization of the threat models is a fundamental step required to minimize the number of false positive and false negative results yielded by a vulnerability analysis technique. This list of threat models is not exhaustive but new attack patterns can be supported by modeling them with their corresponding constraint.

Threat models 1, 12, and 13 are applicable to the input used in element contents of HTML and XML documents. They reflect the attacks containing meta-characters such as < and >, which can be used to inject additional (malicious) elements into a document. For example, the constraint corresponding to the first threat model *input*`.matches(".*[<>/].*")` matches attacks like `<script>alert();</script>`.

Threat models 2, 3, and 17 are applicable to the input used as value of event handlers, for source attributes in HTML documents, and for external entities in XML documents. No input should be allowed in these contexts, since an attack can be conducted by simply providing URLs gointing to malicious hosts, by injecting JavaScript code such as `javascript:alert()`, or by using the value `/etc` in an external entity of an XML document (to gain unauthorized access to local files). Moreover, input sanitization would not help in this case, since these attacks do

TAUTCSTR($input$, $ctxDel$) = TAUTCSTRNUM($input$, $ctxDel$)$\vee$
TAUTCSTRSTR($input$, $ctxDel$)

TAUTCSTRNUM($input$, $ctxDel$) = $\bigvee_{nrel \in \{>,<,\leq,\geq,=,\neq\}}$(
$input$.matches(".*".concat($ctxDel$.concat(" +[Oo][Rr] "))
.concat(N_1.toString()).concat($nrel$.toString())
.concat(N_2.toString()).concat(".*") $\wedge$ N_1 $nrel$ N_2)

TAUTCSTRSTR($input$, $ctxDel$) = $\bigvee_{cstr \in \{=,\neq\}} \bigvee_{d \in \{',"\}}$(
$input$.matches(".*".concat($ctxDel$.concat(" +[Oo][Rr] "))
.concat(d).concat(S_1).concat(d).concat($cstr$.toString())
.concat(d.concat(S_2).concat(d)).concat(".*")) $\wedge$
S_1 $cstr$ S_2)

Figure 2.1: Tautology constraint template TAUTCSTR.

not need to use meta-characters to be effective. Hence, these threat models are expressed with the constraint $input$.matches(".+"), which enforces $input$ to match any character except the empty string.

Threat models 4–7, 14–16, and 21 are applicable to the input used as the value of HTML, XML, and LDAP attributes. The difference among these models lies in the different type of quotation used for the attribute values. For example, if an input is enclosed with single quotes (as in <div attr='$input$'), an attack could be conducted by providing as input the single quote character ' followed by an attack payload (e.g., a payload string like ' onmouseover=javascript:alert(), which would inject an additional JavaScript event). Such an attack is matched by the .*'.* regular expression. A similar threat model is defined for inputs with double quotes. If the input is not enclosed by any type of quote, various meta-characters (e.g., =, *, and ;) may be used to conduct an attack like the one above. This type of attack is matched by threat models 6, 7, and 16, where the list of meta-characters is specific to the context in which they can be applied.

Threat models 8–11 and 18–20 are applicable to the input used as attribute value in SQL and XPath queries. These models reflect the various patterns of tautology attacks discussed in Section 2.1.1, which cause the selection clause of an SQL or XPath query to always evaluate to true. We express them using the parameterized constraint template TAUTCSTR, whose definition is shown in Figure 2.1. This template has two parameters: $input$ is a string variable representing the input to be matched against the tautology pattern; $ctxDel$ represents the string delimiter used for enclosing the context of $input$ (as shown in threat models 8–10 and 18–19). The template is defined as a disjunction of two constraints, each of them expressed through a subtemplate: TAUTCSTRNUM, expressing numeric tautology attacks of the form x' or N_1 $nrel$ N_2, where N_1 and N_2 are integer variables and $nrel \in \{>, <, \leq, \geq, =, \neq\}$; TAUTCSTRSTR, expressing string tautology attacks of the form x' or S_1 $cstr$ S_2, where S_1 and S_2 are string variables and $cstr \in \{=, \neq\}$. Template TAUTCSTRNUM is defined as a disjunction of constraints over $nrel$; each disjunct consists of two conjuncts:

1. The first conjunct generates a pattern against which the user input variable $input$ has to be matched. The concatenation of string ".*" with the context delimiter $ctxDel$ encloses the context of $input$. Afterwards, the actual attack pattern is generated by concatenating the string " +[Oo][Rr] " with the string representation of N_1, together with the string

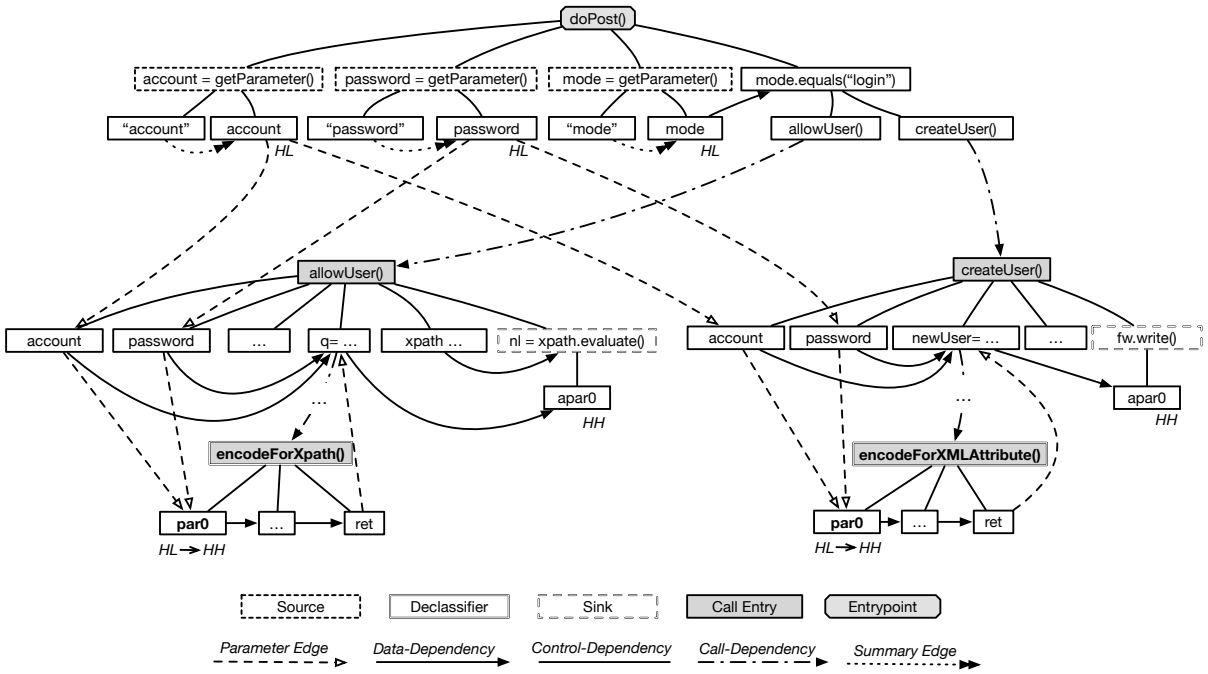


Figure 2.2: The system dependence graph (SDG) of the program in Listing 2.1.

representation of $nrel$, which is then concatenated with the string representation of N_2 and the string `".*"`.

2. The second conjunct $N_1\, nrel\, N_2$ enforces the numeric constraint defined by $nrel$ on N_1 and N_2 to ensure that only satisfiable tautologies are accepted.

TAUTCSTRSTR is defined as a disjunction of constraints over $cstr$; each disjunct consists of two conjuncts, which are structurally similar to those used in the definition of TAUTCSTRNUM. The main difference is that string variables S_1 and S_2, when used in a string concatenation, are always enclosed by a pair of delimited characters represented by the variable d, which ranges (through the inner disjunction) over the set $\{$`'`, `"`$\}$.

2.2 Static Analysis & Program Slicing

Our terminology and definitions regarding static analysis and program slicing are based on those of Hammer [47]. This section provides definitions for *control-flow graphs*, *program dependence graphs*, *system dependence graphs*, *backward program slices*, and *forward program slices*.

Definition 2.2.1 (*Control Flow Graph [38]*) A Control Flow Graph (CFG) is a directed graph $G = (N, E, Start, End)$ N is the set of nodes representing statements and predicates and E is the set of control-flow edges. The control flow graph G is augmented with a unique *entry* node *Start* and a unique *exit* node *End* such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes *T* (true) and *F* (false) associated with the

outgoing edges in the usual way. We further assume that for any node n in N there exists a path from *Start* to n and a path from n to *End*.

Definition 2.2.2 (*Interprocedural Control Flow Graph [47]*) An Interprocedural Control Flow Graph (ICFG) $G = ((G_p)_{p \in P}, \mathit{main}, \mathit{Call}, \mathit{Ret})$ for a program P consists of a family $(G_p)_{p \in P}$ of CFGs $G_p = (N_p, E_p, \mathit{Start}_p, \mathit{End}_p)$ for procedures $p \in P$ an entry procedure *main*, and sets of call and return edges *Call* and *Ret* such that

1. Both $(N_p)_{p \in P}$ and $(E_p)_{p \in P}$ are each pairwise disjoint.
2. If $(u, v) \in \mathit{Call}$, then $u \in N_p \setminus \{\mathit{Start}_p\}$ and $v = \mathit{Start}_{p'}$ for some $p, p' \in P$, and there is a matching return edge $(\mathit{End}_{p'}, u') \in \mathit{Ret}$ such that $u' \in N_p$ is the only successor to u in G_p. p is the caller and p' the callee for that call edge. We say that u and u' match each other and call (u, u') a call-return edge.
3. Conversely, every return edge in *Ret* has a matching call edge in *Call*.

Nodes with outgoing call edges (incoming return edges) are called call nodes (return nodes).

Definition 2.2.3 (*Program Dependence Graph [38]*) A Program Dependence Graph (PDG) is a directed graph $G = (N, E)$, where N is the set of nodes representing the statements of a given procedure in a program, and E is the set of control-dependence and data-dependence edges that induce a partial order on the nodes in N.

Since a PDG can only represent an individual procedure, slicing on an PDG merely results in intraprocedural slices. For computing program slices from interprocedural programs, Horwitz et al. [52] defined system dependence graphs, which are essentially interprocedural program dependence graphs from which *interprocedural* program slices can be soundly and efficiently computed.

Definition 2.2.4 (*System Dependence Graph [52]*) A System Dependence Graph (SDG) consists of all the PDGs in the program, which are connected using interprocedural edges that reflect calls between procedures. This means that each procedure in a program is represented by a PDG. The PDG is modified to contain *formal-in* and *formal-out* nodes for every formal parameter of the procedure. Each call-site in the PDG is also modified to contain *actual-in* and *actual-out* nodes for each actual parameter. The call node is connected to the entry node of the invoked procedure via a *call* edge. The *actual-in* nodes are connected to their corresponding *formal-in* nodes via *parameter-in* edges, and the *actual-out* nodes are connected to their corresponding *formal-out* nodes via *parameter-out* edges. Lastly, *summary edges* are inserted between *actual-in* and *actual-out* nodes of the same call-site to reflect transitive data-dependencies that may occur in the called procedure.

Since an SDG provides an interprocedural model of a program—capturing interprocedural data-dependencies, control-dependencies, and call-dependencies—it is the ideal data structure for program analysis. Furthermore, program slices can be computed from it in a sound and efficient way in linear time [52, 93]. More specifically, the worst-case complexity of building a program slice from an SDG of N nodes is $O(N)$; the worst-case complexity of building an SDG itself is $O(N^3)$ [47].

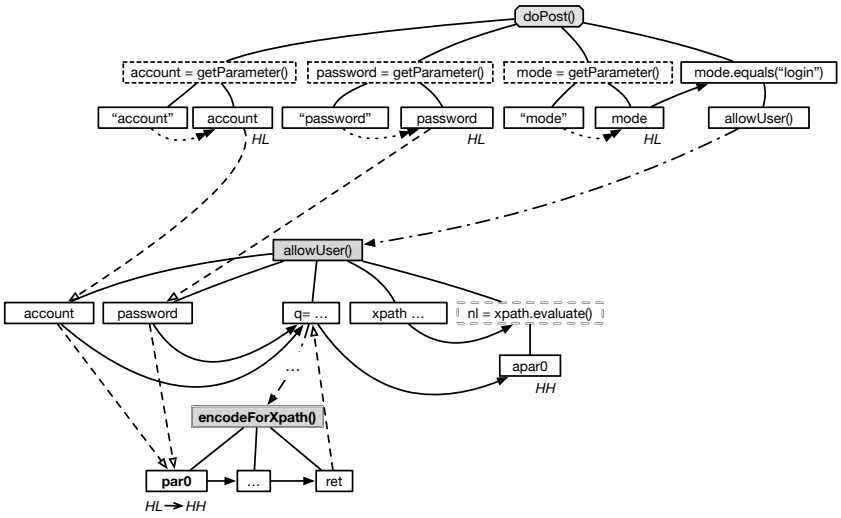


Figure 2.3: The backward program slice with respect to the sink at Line 18 in Listing 2.1.

Figure 2.2 depicts the SDG of the program in Listing 2.1. The entry points of the methods `allowUser`, `createUser`, `encodeForXpath` and `encodeForXMLAttribute` as well as the main entry point `doPost` are represented as SDG nodes (shaded boxes). The other nodes (white boxes), which represent the expressions of the program in Listing 2.1, are connected with control-dependence edges (black lines), data-dependence edges (black arrows) and summary edges (dotted black arrows). Call edges (dashed arrows with black arrowheads) connect call sites with their respective targets, whereas dashed arrows with white arrowheads denote parameter edges. Input sources are highlighted with a solid dashed frame, whereas sinks are highlighted with a blank dashed frame.

Definition 2.2.5 (*Backward Program Slice [52]*) Given an SDG $G = (N, E)$, let $K \subseteq N$ be the set of identified sinks. The backward program slice of G with respect to a target criterion $k \in K$, denoted with $bs(k)$, consists of all the statements that influence k, and is defined as $bs(k) = \{j \in N \mid j \xrightarrow{*} k\}$, where $j \xrightarrow{*} k$ denotes that there exists an *interprocedurally-realizable path* from j to k, so that k is reachable through a set of preceding statements (possibly across procedures). The detailed algorithms for computing interprocedurally-realizable paths and backward slice are given in [52].

As illustrated in Figure 2.3, the backward program slice with respect to the sink at Line 18 in Listing 2.1 contains all the program statements that influence (both intraprocedurally and interprocedurally) the operation of the sink.

Definition 2.2.6 (*Forward Program Slice [20]*) Given an SDG $G = (N, E)$, let $I \subseteq N$ be the source criterion. The forward program slice of G with respect to I consists of all the nodes that are influenced by I, and is defined as $fs(I) = \{j \in N \mid i \xrightarrow{*} j \wedge i \in I\}$.

The program in Listing 2.1 contains three input sources at Lines 2–4; Figure 2.4 shows the forward program slice with respect to the input source `account` at Line 2.

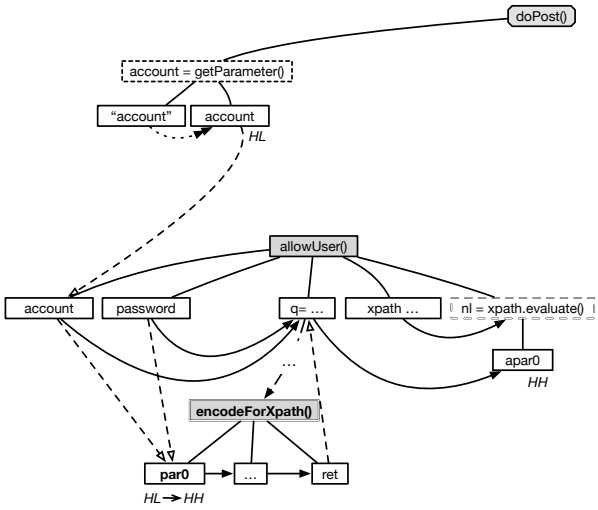


Figure 2.4: forward slice with respect to the slicing criterion at Line 2 in Listing 2.1.

Definition 2.2.7 (*Program Chop [57, 105]*) The program chop of an SDG $G = (N, E)$ with the source criterion I and the target criterion k is defined as $c(I, k) = bs(k) \cap fs(I)$.

Note that program chopping is defined as the intersection of backward slicing and forward slicing. It allows us to identify security-relevant nodes that are on the paths from I to k and, thus, involved in the propagation of potentially malicious data from input sources to a sink.

For example, Figure 2.5 shows a chop between the input sources `getParameter` on Lines 2–4 and the sink `xpath.evaluate` on Line 18.

2.3 Constraints and Constraint Networks

This section provides definitions for *constraint* and *constraint network*.

Let $Y = y_1, \ldots, y_k, k > 0$ be a finite sequence of variables and $D_1, \ldots, D_k$ a sequence of domains, with each variable y_i ranging over the respective domain D_i.

Definition 2.3.1 (*Constraint*) A constraint c over Y is a relation over Y, i.e., $c \subseteq D_1 \times \cdots \times D_k$; Y is also called the scope of the constraint and k is its arity. Informally, a constraint c on some variables is a subset of the cartesian product over the variables domains that contains the combination of values that satisfy c.

Definition 2.3.2 (*Constraint Network*) A constraint network $\mathcal{R}$ is a triple (X, D, C), where X is a finite sequence of variables $x_1, \ldots, x_n$, each associated with a domain $D_1, \ldots, D_n$ and $C = \{c_1, \ldots, c_t\}$ is a set of constraints; the scope of each constraint c_i, denoted with S_i, is a subsequence of X.

A constraint network $\mathcal{R} = (X, D, C)$ can be represented as a *hypergraph* $H = (V, S)$ where the set of nodes V corresponds to the set of variables X of the network, and $S = S_i, \ldots, S_t$ is the set of hyperedges that group variables belonging to the same scope.

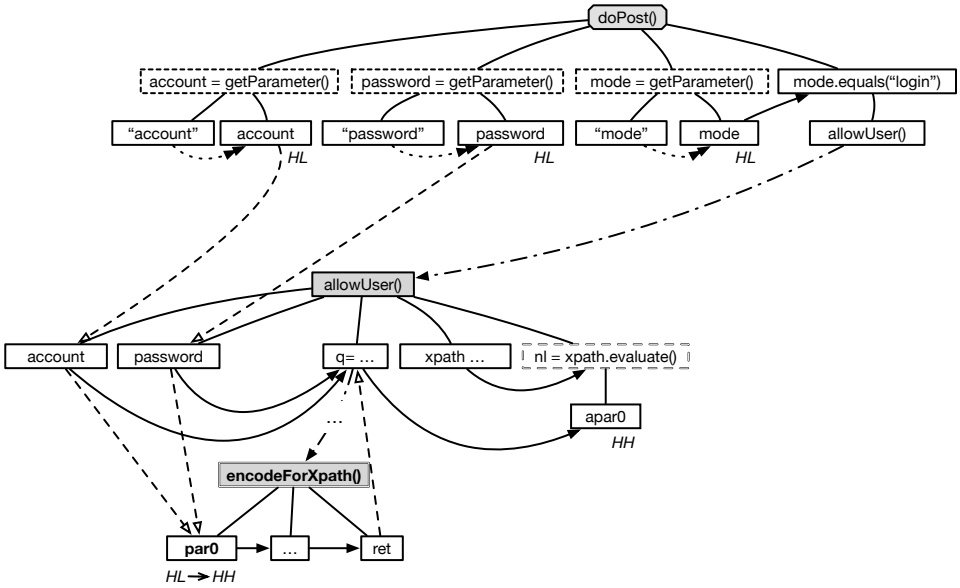


Figure 2.5: The chop with the source criterion {2, 3, 4} and the target criterion {18} of the example program in Listing 2.1.

2.4 Ant Colony Optimization

Ant Colony Optimization (ACO) [37] is a widely used meta-heuristic search technique for solving combinatorial optimization problems. It has been applied to a wide variety of optimization problems such as scheduling and routing [25], data-compression [19] or user interface generation [16].

ACO is inspired by the observation of the behavior of real ants searching for food. Real ants start seeking food randomly; when they find a source of food, they leave a chemical substance (called *pheromone*) along the path that goes from the food source back to the colony. Other ants of the colony can detect the presence of this substance and are likely to follow the same path. This path, populated by many ants, is called *pheromone trail* and serves as a guidance (e.g., positive feedback) for the other ants. Notice that the shorter the path, the sooner the pheromone is deposited along it and the more ants use it. When the source of food is depleted, the path is no longer populated by ants and the pheromone evaporates: the ants forget this path and start exploring other search directions. These observations can be translated into the world of *artificial ants*, which can cooperate to find a good solution to a given optimization problem.

The optimization problem is translated into the problem of finding the best path on a weighted graph. *Artificial pheromone trails* are numerical parameters that characterize the graph components (i.e., vertices and edges). Artificial pheromone trails can be read/written by ants and represent the sole means of communication among the ants; these trails encode the "history" in approaching the problem (and finding its solutions) by the whole ant colony. ACO algorithms also implement a mechanism, inspired by real pheromone evaporation, to modify the pheromone information over time so that ants can forget the (search) history and start exploring new search directions. The artificial ants build their solutions by moving step-by-step along

the graph; at each step they make a stochastic decision based on the pheromone trail information.

To apply the ACO meta-heuristic to a combinatorial optimization problem, first one has to define the optimization problem as $P = (S, \Omega, f)$, where S is the search space defined over a finite set of discrete decision variables; Ω is a set of constraints among the variables; $f : S \to \mathbb{R}_0^+$ is the objective function to minimize. The search space S is defined as follows: given a set of discrete variables $X_i, i = 1 \dots n$, with values $v_i^j \in D_i = \{v_i^1, \dots, v_n^{|D_i|}\}$, a feasible solution $s \in S$ is a complete assignment in which each variable is assigned a value in its domain such that all the constraints in Ω are satisfied. A feasible solution $s^* \in S$ is called a global minimum of P if and only if $f(s^*) \leq f(s)\ \forall s \in S$; S^* is the set of global minima.

This model of an optimization problem can be used to derive a generic pheromone model for use by ACO. Let us denote with c_{ij} a solution component, i.e., an instantiation of a decision variable X_i with a value $v_i^j \in D_i$; solution components are combined by ants to form feasible solutions. Each solution component c_{ij} has an associated pheromone trail parameter T_{ij}. Let us denote the set of all solution components as C and the set of all pheromone trail parameters as T. Each pheromone trail T_{ij} has a pheromone value τ_{ij}, which indicates the desirability of choosing the corresponding solution component; this pheromone value is read and written by the ACO algorithm during the search.

Artificial ants build a solution to an optimization problem by traversing the *construction graph* $G_C(V, E)$, where V is a set of vertices and E is a set of edges. The construction graph G_C is defined by associating the set of solution components C either with V or with E. The ants move from one vertex to another along the edges of the construction graph, based on the pheromone values; in this way, they incrementally build a solution. During each move, the ants deposit a $\Delta\tau$ amount of pheromone either on the vertices or on the edges they traverse; this amount of pheromone depends on the quality of the solution found. The pseudocode of the ACO meta-heuristic is shown in Algorithm 2.1. After initializing the parameters and the pheromone trails, the algorithm loops through three main steps, until the termination conditions are met:

1. *ConstructAntSolutions*: this step builds solutions from the set of available solution components C. Each solution component is selected probabilistically, using a heuristic function that takes into account the pheromone trail.

2. *ApplyLocalSearch*: this optional step is used to refine the set of candidate solutions built in the previous step, to locally optimize them.

3. *UpdatePheromones*: this step updates the pheromone values of solution components. The update can increase or decrease (to emulate pheromone evaporation) the pheromone value, depending on whether the corresponding solution component belongs to a good or bad solution. The quality of each solution is assessed by means of a *fitness function*.

```
function ACO
    initialize parameters and pheromone trails
    while termination conditions are not met do
        ConstructAntSolutions
        ApplyLocalSearch
        UpdatePheromones
    end while
end function
```

Algorithm 2.1: Ant Colony Optimization (ACO) meta-heuristic.

Part II

A Hybrid Analysis Framework

Chapter 3

Security Slicing

The chapter introduces our security slicing approach and is organized as follows: Section 3.1 motivates security slicing; Section 3.2 explains all the security slicing steps; Section 3.3 explains the implementation of our prototype tool *JoanAudit*; Section 3.4 reports on the evaluation results; Section 3.5 concludes this chapter.

3.1 Overview

Software security assurance is an important process in software development that protects the sensitive data and resources contained in and controlled by the software. Addressing security vulnerabilities early in the software development stage could decrease the cost of addressing them in later stages by a factor ranging between 30 and 60 times [84].

Security auditing, i.e., the examination of the source code for the purpose of detecting vulnerabilities, helps to detect vulnerabilities during the early phases of software development. However, without proper automation, this task is laborious, error-prone and not scalable; hence, security auditors need automated support to facilitate the auditing process.

In order to support security auditors with their tasks, auditing approaches face the following challenges [131]:

C1 they have to help auditors locate the vulnerabilities quickly in the source code.

C2 they need to scale to the size of realistic Web systems.

C3 they should generate reports that provide control-dependency information to detail how injection vulnerabilities reach the sink in order to eliminate false alarms quickly.

C4 these reports should only provide information relevant to security.

C5 they need to support various types of vulnerabilities, such as XSS, SQLi, XMLi, XPathi or LDAPi.

C6 they need to support security analysis for the Java programming language, one of the most commonly used technologies for Web development in industrial context [26].

C1 and *C2* are addressed by approaches based on taint analysis [78, 60, 138, 98, 137, 53, 71]. However, reports generated by these approaches typically contain data-flow analysis traces and lack *control-dependency information* (*C3*), which is essential for security auditing. Indeed, conditional statements checks are often used to perform input validation or sanitization tasks; without analyzing such conditions, feasible and infeasible data-flows cannot be determined, causing many false warnings.

C3 is addressed by approaches based on symbolic execution [64, 153] which helps to identify and locate potential vulnerabilities in program code, and thus, could assist the auditor's tasks. Though symbolic execution approaches reason with control-dependency information, they have yet to address scalability issues (*C2*) due to the path explosion problem [146]. Other approaches [144] report analysis results without any form of pruning (*C1* and *C4*), thus containing a significant amount of information not useful to security auditing. As a result, an auditor might end up checking large chunks of code, which is not practical.

C5 is also not addressed by the majority of the above-mentioned approaches; the only exception is [98], which explicitly addresses XMLi, XPathi, and LDAPi.

Challenges *C2*, *C4*, and *C5* are addressed by security testing approaches [5, 58, 10, 69, 129] and dynamic analysis-based security attack detection approaches [80, 106, 101, 123, 46, 116, 124]. These approaches can be used to detect XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. However, a security auditor is typically required to locate vulnerabilities in source code (*C1*), identify their causes and fix them. Analysis reports from the above-mentioned approaches, though useful, are not sufficient to support code auditing since they only contain information derived from observed program behaviors or execution traces; they do not provide information about the location of the vulnerability in the source code.

C6 is generally addressed by black-box security testing approaches [10, 58, 129] because they are agnostic with respect to the programming language of the system under test. However, this is the same reason for which these approaches cannot locate vulnerabilities in the source code (*C1*). Some security testing based approaches [69, 46] and static-analysis approaches [98, 53] do support Java but cannot meet *C1* and *C3*, respectively.

In this chapter, we present *security slicing*, a technique that facilitates security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities in program source code. In this approach we first apply static analysis to identify the *input sources* and the sinks; afterwards, we apply program slicing and *code filtering* techniques to extract minimal and relevant source code that contains only statements required for auditing potential vulnerabilities related to each sink, pruning away other statements that do not require auditing. Security slicing addresses all of the above-mentioned challenges by generating a vulnerability report that locates the vulnerabilities in the source code (*C1*), being scalable to Web systems realistic in size (52 kLOC) (*C2*), extracting control- and data-dependency information from the program (*C3*), generating precise security slices that do not miss security-relevant information (*C4*), being readily configured for common injection vulnerabilities (*C5*), and targeting Java Web systems (*C6*).

```
1:  function SECSLICE(
    a program W
    Set of irrelevant/known-good library methods M_⟨IR,KG⟩
    Set of sources, sinks and declassifiers Λ_⟨I,K,D⟩)
2:      SS ← ∅                    ▷ Set of security slices and associated path conditions
3:      SDG g ← COMPUTESDG(W)
4:      g' ← PRUNE(g, M_⟨IR,KG⟩)                              ▷ Apply filter 1 and 2
5:      ⟨I, K⟩ ← GETSRC-SNK(g', Λ_⟨I,K,D⟩)
6:      for all k ∈ K do
7:          c(I, k) ←CHOP(g', I, k)                           ▷ Apply filter 3
8:          ss(I, k) ← IFCANALYSIS(c(I, k))                   ▷ Apply filter 4
9:          ⟨ss(I, k)', PC⟩ ← CONTEXTANALYSIS(ss(I, k))       ▷ Apply filter 5
10:         SS ← SS ∪ {⟨ss(I, k)', PC⟩}
11:     end for
12: end function
```

Algorithm 3.2: Security slicing algorithm.

3.2 Approach

This section illustrates the security slicing approach in detail: Section 3.2.1 presents the security slicing algorithm, and Section 3.2.2 explains the different security slicing steps; Section 3.2.3 illustrates the application of security slicing on an example Web application.

3.2.1 Algorithm

Our fully-automated approach mainly targets Java-based Web applications, since the type of vulnerabilities it supports are commonplace in such systems. We emphasize that a specialized approach is necessary to provide practical support for the security auditing of Web applications and services developed using a specific technology.

When extracting security slices, we aim to achieve the following objectives:

1. *Soundness:* A security slice shall contain all the relevant program statements enabling the auditing of any security violation.

2. *Precision:* A security slice shall contain only the program statements relevant to minimizing the auditing effort.

3. *Performance:* The security slicing algorithm shall handle Web applications of realistic size.

Achieving all these objectives is desirable but in practice there is a trade-off between soundness and precision, depending on the analysis goal. In our context, we prioritize soundness because finding all the possible security violations is a priority for security auditing; nevertheless, we also try to optimize precision to the extent possible.

The pseudocode of the algorithm realizing our security slicing approach is shown in Algorithm 3.2. The algorithm takes as input: the bytecode W of a Java program; a set $M_{\langle IR,KG\rangle}$ of methods (custom functions or library API) that are either irrelevant to security analysis of XSS, SQLi, XMLi, XPathi, and LDAPi, or that may be relevant to security but are known (or assumed) to be correct or free from security issues; a set $\Lambda_{\langle I,K,D\rangle}$ of sources, sinks, and declassifiers (nodes

in the SDG that represent sanitization procedures). The algorithm returns the set SS of security slices and associated path conditions extracted from W.

The algorithm works as follows. After initializing SS to the empty set, it constructs the SDG from the bytecode W of the input program; this step is realized by using the API of *Joana* [47]. The resulting SDG is then filtered by pruning nodes that contain methods belonging to $M_{\langle IR,KG\rangle}$; the details of this step are described in Section 3.2.2.3. The next step identifies the set of input sources I and sinks K from the SDG. Afterwards, the algorithm iterates through the set K; for all sinks $k \in K$, it performs the following steps:

1. Computing the program chop $c(I, k)$, to extract the program slice that contains the statements influenced by the set of input sources I, which lead to sink k through possibly different program paths. This step is realized using the API of *Joana*.

2. Performing Information Flow Control (IFC) analysis to identify how insecure the information flows along the paths in $c(I, k)$ are. This step, partially supported by *Joana*, is described in Section 3.2.2.1.

3. Performing symbolic execution and context analysis to identify the context of sink and to understand whether input data is used in an insecure way in a sink. This analysis automatically patches vulnerable sinks with sanitization procedures if it is able to identify adequate procedures from the extracted path conditions PC. If this is not possible, the extracted information can still be used to facilitate manual security auditing (e.g., checking feasible conditions for security attacks). This step is detailed in Section 3.2.2.2.

Each of the last three steps is combined with a filtering procedure, based on the extracted information flow traces and path conditions; the filtering procedures are explained in Section 3.2.2.3. Furthermore, each iteration terminates by computing a *security slice* $ss(I, k)$ and its path conditions PC, which are then added to set SS.

3.2.2 Detailed Steps

In the following we are illustrating the 3 main steps of security slicing: Section 3.2.2.1 provides a detailed explanation about IFC analysis; Section 3.2.2.2 introduces symbolic execution and context analysis, and Section 3.2.2.3 explains the various filtering techniques for extracting minimal and concise security slices.

3.2.2.1 Information Flow Control Analysis

Information Flow Control Analysis (IFC) analysis is a technique that checks whether a software system conforms to a security specification. Relying on the work of Hammer [47], we adapt his generic flow-, context-, and object-sensitive interprocedural IFC analysis framework to suit our specific information flow problem with respect to XSS, SQLi, XMLi, XPathi, and LDAPi. Our goal is to trace how information from an input source can reach a sink, and then to analyze which paths in the chops are secure and which ones may not be secure.

We specify allowed and disallowed information flow based on a lattice called *security lattice*, i.e., a partial-ordered set that expresses the relation between different security levels. We use the standard diamond lattice $\mathcal{L}_{LH}$ [88], depicted in Figure 3.1, which expresses the relation between

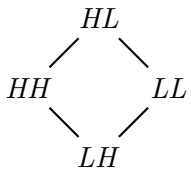


Figure 3.1: The security lattice used in our information flow control analysis.

four security levels HL, HH, LL, and LH. Every level $l = L_0L_1$ contains two components: L_0 denotes the *confidentiality* level while L_1 denotes the *integrity* level. Confidentiality requires that information is to be prevented from flowing into inappropriate destinations or sinks, whereas integrity requires that information is to be prevented from flowing from inappropriate input sources [108]. The element HL represents the most restricted usage, since any data labeled with it cannot flow to any destination that has a different security label. Data labeled with HH are confidential and cannot be manipulated by an attacker, whereas data labeled with LH are non-confidential and also cannot be manipulated by an attacker. The LL label is used for data that are non-confidential but could be altered by an attacker.

All input sources and sinks are annotated with a security label that enables the detection of allowed and disallowed information flow. This annotation step is done automatically based on our predefined sets of input sources and sinks (see Section 2.1.2). Input sources are labeled with HL because data originating from them are supposed to be confidential but could be manipulated by an attacker. Sinks are labeled either with LH or with HH. The value of the confidentiality label is either L or H, depending on whether the sink is allowed or not to handle user confidential data. In any case, the integrity label for sinks is always H, because only high-integrity data should be allowed to flow into the sinks, to prevent the flow of malicious input values causing security attacks. More specifically, in our approach we label as HH the sink functions that update or modify databases—since it is common to store highly-confidential data in back-end databases—as well as the functions that access server environment variables, read data from configuration files or other sources. Moreover, we label as LH the sink functions that generate outputs to external environments, such as exception handling functions, as well as functions that read time and date such as `getTime` from `java.util.Calendar`. Finally, an example of function labeled with LL is a function that monitors mouse-clicks.

Based on these annotations, the IFC analysis traces information flow from one node in the chop to another and detects disallowed information flow and, therefore, security violations. For example, a security violation is detected if there exists an information flow from an LL input source to an HH sink.

Notice that the annotation procedure must also take into account the fact that program developers might use sanitization procedures to properly validate data from an input source before using it in a sink. For instance, this is the case for our example in Listing 2.1, where proper sanitization procedures (Lines 16–17 and Lines 22–23) taken from the OWASP security library [94] are used between the input sources and a sink. Such cases can be considered secure and do not need to be reported to an auditor. To support the use of these functions, we rely on the concept of declassification [109]. In our context, *declassifiers* are nodes in the SDG that represent sanitization procedures. The integrity level of such nodes is annotated with an H label since the sanitization procedure ensures the integrity of data. As we address five different

vulnerability types, only the declassifiers relevant to the vulnerability type of a sink k are annotated with the integrity level H. Other declassifiers in the chop $c(I, k)$ and irrelevant for the vulnerability type of k are ignored. For example, the declassifier at Lines 16–17 in Listing 2.1 is relevant for the XPath function `xpath.evaluate` at Line 18, but is inappropriate for a sink of a different vulnerability category, e.g., an SQL query operation.

In addition to annotating the integrity level of declassifier nodes with H, we also change the integrity level of *the data that reach these nodes* to H. For example, as shown in Figure 2.2, the input sources `account` and `password` (Lines 2–3 in Listing 2.1) are annotated with the label HL. Since these input values pass through the declassifiers at Lines 16–17 (highlighted in bold in Figure 2.2), their security labels are changed to HH. When performing IFC analysis, the use of these variables in the sink node `xpath.evaluate` at Line 18 will be considered secure, because the information flow from HH to HH is allowed. Our tool is configured with the declassifiers (mainly encoding and escaping functions) from two widely-used security libraries—Apache Common [8] and OWASP [94]. It also recognizes the `PreparedStatement` function from the `java.sql` package as a declassifier corresponding to SQL sinks.

Consider now the same example above, but without sanitization functions. In such a case, we would have at least two illegal flows (from `account` and `password` to the `xpath.evaluate` sink) from HL to HH. Hence, their corresponding paths would be determined as potentially insecure and will be subject to *context analysis*, explained in the next subsection.

3.2.2.2 Symbolic Execution & Context Analysis

The IFC analysis illustrated in the last section can tell *if* data from input sources may reach sinks. However, from a security auditing standpoint it is also necessary to understand the *context* of a sink, i.e., *how* the input data is used in a sink and *if* it is used in an insecure way.

In this section, we present *context analysis*, a lightweight technique for identifying the context (within a sink) in which the data of an input source is used. Based on the identified context, this technique is able to automatically fix a vulnerable input source by applying the most appropriate sanitization function to it.

Table 3.1 lists, for each type of vulnerability that we consider, the possible contexts (in the form of patterns, where **input** correspond to the data from an input source). For each context, we indicate[1] the most appropriate security API (provided by OWASP [94]) that should be used in that specific context to sanitize the input data.

Context analysis is lightweight compared to symbolic evaluation and constraint solving approaches [64, 153] because it traverses only the paths leading to the sink rather than the whole program, and does not attempt to precisely reason about the operations performed in the path (e.g., by performing constraint solving). Instead, the analysis merely collects and examines the path conditions, i.e., the necessary conditions for the presence of information flow from input sources I to a sink k via a program path. More specifically, context analysis relies on path condition analysis to rule out infeasible paths, and to reconstruct the string values in the sink, needed to identify the context of the input source. The identified context is matched with the context patterns of Table 3.1. In case of a match, context analysis applies the corresponding fix, by wrapping the input source causing the vulnerability with the proper security API. Otherwise, in case

[1]Table 3.1 shows the mapping between context patterns and security APIs as configured in our tool. Nevertheless, users can provide a different mapping.

Table 3.1: Mapping between contexts and security APIs for data sanitization.

Vul.	Nr.	Context Pattern	Security API
XSS	1	HTML element content: <tag>**input** </tag>	ESAPI.encoder().encodeForHTML
	2	HTML attribute value: 	ESAPI.encoder().encodeForHTMLAttribute
	3	URL parameter value: <a href="http://...?param=**input**">	ESAPI.encoder().encodeForURL
	4	JavaScript variable value: <script>var a='**input**'...</script> ...	ESAPI.encoder().encodeForJavaScript
	5	CSS property value: <style>selector {property:**input**;}</style> ...	ESAPI.encoder().encodeForCSS
SQLi	6	SQL attribute value: SELECT column From table WHERE row='**input**'	ESAPI.encoder().encodeForSQL
XMLi	7	XML element content: <node>**input**</node>	ESAPI.encoder().encodeForXML
	8	CDATA content: <![CDATA[**input**]]>	ESAPI.encoder().encodeForXML
	9	XML attribute value: <node attr='**input**'/>	ESAPI.encoder().encodeForXMLAttribute
XPathi	10	XPath attribute value: //table[column='**input**']	ESAPI.encoder().encodeForXPath
LDAPi	11	LDAP distinguished name: LdapName dn = new LdapName(**input**)	ESAPI.encoder().encodeForDN
	12	LDAP search: search="(attr=**input**)"	ESAPI.encoder().encodeForLDAP

there is no match and the input source cannot be fixed automatically, the procedure yields the path conditions, which represent a valuable asset for security analysts to understand the cause of a vulnerability.

To explain this analysis, we use the code snippet shown in Figure 3.2 and extracted from one of our test subjects *WebGoat*/`MultiLevelLogin1` (see Section 3.4). The code is vulnerable to XSS because the input data, which is accessed from a database (source at Line 15) and displayed as content of an HTML page (sink at Line 30), could be tampered with by an attacker before the data is stored in the database.

Context analysis uses SSA form [33], a standard intermediate representation used in program analysis. In SSA form, every variable in a program is assigned exactly once and every variable is defined before it is used. For join points, i.e., points in the program where different control flow paths merge together, a Φ-operation is added to represent the different values that a variable can take at that point. Figure 3.2b shows the equivalent SSA form for the program in Figure 3.2a.

The pseudocode of our context analysis function is shown in Algorithm 3.3. It takes as input a security slice ss in a dependence graph form; it uses two local variables: PC, representing the set of preconditions analyzed, and P_V, representing the set of potentially vulnerable paths.

First, the input security slice ss is transformed by function GENICFG into its equivalent ICFG form [119], which shows the order of control flow executions across procedures. In this form, the control flow paths in the slice become explicit and can be easily extracted.

Afterwards, function COLLECTPATHS extracts the control flow paths by traversing the ICFG

```
1 String q = "SELECT * FROM msg " +
2     WHERE usr LIKE ?";
3 String out = "<html>";
4 Connection c = DriverManager
5     .getConnection(DB);
6 PreparedStatement s = c
7     .prepareStatement(q);
8 s.setString(1, getUser());
9 ResultSet r = s.executeQuery();
10 int i = 0;
11
12 while (r.next()){
13
14
15   String u = r.getString(1); // SOURCE
16
17   if (!u.isEmpty()) {
18     out += "<p>" + i + " " +
19       u.toUpperCase() + "</p>";
20
21
22
23
24
25   }
26   i++;
27
28 }
29 out += "</html>";
30 println(out); // SINK
```

(a) A Java program.

```
1 String q1 = "SELECT * FROM msg " +
2       WHERE usr LIKE ?";
3 String out1 = "<html>";
4 Connection c1 = DriverManager
5     .getConnection(DB);
6 PreparedStatement s1 = c1
7     .prepareStatement(q1);
8 s1.setString(1, getUser());
9 ResultSet r1 = s1.executeQuery();
10 int i1 = 0;
11 boolean t1 = r1.next();
12 while [i2 = Φ(i1,i3) ,
13 out2 = Φ(out1,out8),
14 t2 = Φ(t1,t3)] (t2) {
15   String u1 = r1.getString(1); // SOURCE
16   boolean k1 = u1.isEmpty();
17   if (!k1) {
18     u2 = u1.toUpperCase();
19     out3 = out2 + "<p>";
20     out4 = out3 + i2;
21     out5 = out4 + " ";
22     out6 = out5 + u2;
23     out7 = out6 + "</p>";
24   }
25   out8 = Φ(out2,out7)
26   i3 = i2 + 1;
27   t3 = r1.next();
28 }
29 out8 = out2 + "</html>";
30 println(out8); // SINK
```

(b) A Java program in SSA form.

Figure 3.2: The Java source code (a) and the equivalent SSA form of a sample program (b).

in a depth-first search manner. For practicability (to avoid path explosion), loops and recursive function calls are traversed only once; both our experience and the evidence gathered during our experiments confirm that analyzing one iteration of loops and recursive calls is sufficient to detect vulnerabilities. To illustrate this step, we use the ICFG of the program from Figure 3.2b, shown in Figure 3.3. Every control flow edge is labeled with a sequence number; outgoing predicate edges are annotated with *TRUE* or *FALSE*. In the figure, three control flow paths can be observed: $\{(1,8),(1,2,3,6,7,8),(1,2,3,4,5,7,8)\}$. However, for this program, the IFC analysis described in Section 3.2.2.1 would have already pruned the paths $\{(1,8),(1,2,3,6,7,8)\}$ from the security slice, since there is no insecure information flow in those paths. Hence, function COLLECTPATHS will return, in variable P_V, only one potentially vulnerable path: $P_V = \{(1,2,3,4,5,7,8)\}$.

The next step of the context analysis procedure is a loop that iterates over the set P_V. For each path $p \in P_V$, function EVALPATH tries to automatically fix the vulnerability contained in p, if possible. Function EVALPATH, which takes in input a path p, works as follows. First, the path conditions pc and the context of the input source ctx of path p are extracted with the EVAL procedure, described further below. Afterwards, function AUTOFIX identifies the required sanitization procedure by matching the extracted context ctx against one of the context patterns shown in Table 3.1. If there is a match for ctx, the security API corresponding to the matched

```
1:  function CONTEXTANALYSIS(Security Slice ss)
2:      PC ← ∅
3:      P_V ← ∅
4:      cfg ← GENICFG(ss)
5:      P_V ←COLLECTPATHS(cfg)
6:      for all p ∈ P_V do
7:          pc ← EVALPATH(p)
8:          if pc ≠ null then
9:              PC ← PC ∪ pc
10:         end if
11:     end for
12:     return ⟨ss, PC⟩
13: end function

14: function EVALPATH(Path p)
15:     ⟨ctx, pc⟩ ← EVAL(p)
16:     fix ←AUTOFIX(ctx)
17:     if fix then
18:         REMOVEPATH(p, ss)
19:         return null
20:     end if
21:     return pc
22: end function

23: function EVAL(Path p)
24:     ⟨Vmap, Cond⟩ ←TRACEBACKWARDS(p)
25:     Vmap′ ←RESOLVEVARIABLES(Vmap)
26:     ⟨srcpar, snkpar⟩ ←GETSRCSNKPARAMS(Vmap′)
27:     return ⟨GETCONTEXT(⟨srcpar, snkpar⟩), ⋀_{c∈Cond}⟩
28: end function
```

Algorithm 3.3: Context analysis algorithm.

context pattern is applied to the input source; this automated fixing procedure is further explained in Section 3.2.2.3. If function AUTOFIX returns a fix, procedure REMOVEPATH is invoked to prune the fixed path from the security slice ss, and EVALPATH terminates returning null. If fixing the vulnerability in p is not possible, the EVALPATH function returns the path condition pc corresponding to path p. The path conditions returned after executing the loop over P_V are available in the set PC, which can be used by security auditors for manual inspection.

The extraction of the path conditions and of the context of a path is done through function EVAL, which works as follows. It traces, in reverse control-flow order starting from the sink, all the statements (in the SSA form) on which the sink variable is data- or control-dependent. Function TRACEBACKWARDS collects all the variables, their assignments and their interdependencies (stored in the map $Vmap$), including the conditions $Cond$ imposed on the variables at *predicate* statements. Function RESOLVEVARIABLES resolves all variables until a fixed point is reached; the variables used in the sink are resolved as a concatenation of the program-defined values and the input variables. The result of the fix-point iteration is stored in the map $Vmap'$, which is then used by the GETSRCSNKPARAMS function to determine: 1) the variables that are associated with the input source $srcpar$, i.e., the value that is returned by the source operation; 2) the sink parameter $snkpar$, i.e., the string that is passed to the operation in the sink. With this information, function GETCONTEXT extracts the context of the input source with respect to the sink. The context is returned together with the conjoined conditions in $Cond$ to the EVALPATH procedure and stored in variables ctx and pc.

For example, after applying the EVAL procedure on the path $p = (1, 2, 3, 4, 5, 7, 8)$ in Figure 3.3, variable $\texttt{out}_8$ at the sink at Line 30 is resolved to:

$$\texttt{<html><p>0 } \mathbf{u_1}\texttt{.toUpperCase()</p></html>}$$

where $\mathbf{u_1}$ represents the input variable assigned with the data from the input source at Line 15. By matching this context against the context patterns of Table 3.1, it is identified as an input used as *the content of an HTML element*. The corresponding security API `ESAPI.encoder().encodeForHTML` is then used to patch the input source at Line 15 in Figure 3.2, resulting in the new statement:

```
String u = ESAPI.encoder().encodeForHTML(r.getString(1))
```

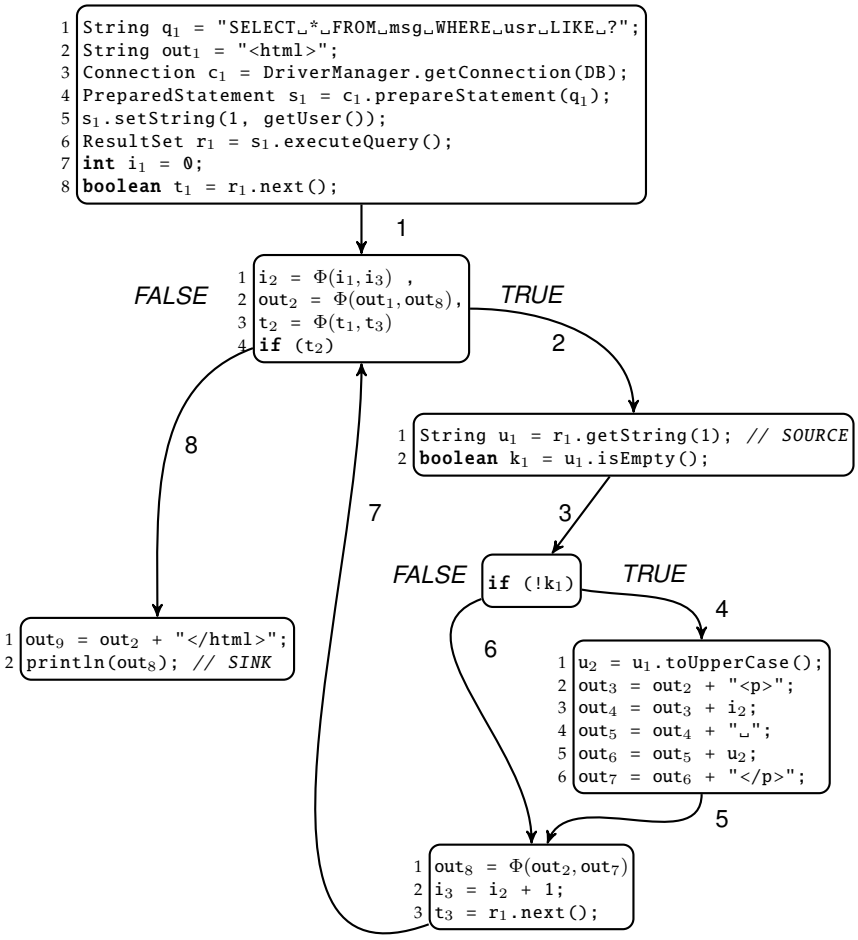


Figure 3.3: The control flow graph of a program in Figure 3.2.

Consider now the case in which the above vulnerable path $p = (1, 2, 3, 4, 5, 7, 8)$ could not be fixed by function AUTOFIX. The following path condition pc would be reported:

```
DriverManager.getConnection(DB).prepareStatement("SELECT...")
.executeQuery().next() ∧ ¬u.isEmpty()
```

Based on this information, a security auditor may easily identify that the path is feasible as long as there are user data in the database. Hence, she may conclude that a security attack is feasible since there is no sanitization of the user input.

Note that our approach filters *known-good* classes (explained in the next subsection) such as those belonging to database drivers and database queries from the SDG. During SDG construction, those classes are replaced with stub nodes. Therefore, for the example above, the paths in the methods called by the `DriverManager` are not explored in our analysis. The considerable reduction of the number of analyzed path improves the scalability of our approach, and results in a simplified path condition, from which an auditor can still assess its feasibility.

3.2.2.3 Filtering

In this section, we describe the five filtering mechanisms (*F1–F5*) which are applied to generate minimal slices for security auditing. For efficiency reasons, the filters are applied at different stages of our approach (as shown in our security slicing algorithm in Algorithm 3.2). *F1* and *F2* are applied concurrently during the SDG construction. *F3* is applied during program chopping. *F4* and *F5* are applied to the program chops in sequence. We mentioned earlier that the goal of our work is to achieve the highest possible *precision* while preserving *soundness* so that security auditing is scalable.

The original program chops $c(I, k)$ without filters are *sound* with respect to the types of input sources and sinks we consider, since all the statements related to those sources and sinks are extracted. It is straightforward to prove that by applying the filtering rules illustrated below, which remove statements that cannot be relevant to security auditing, we achieve better precision compared to the original program chops. However, we also need to demonstrate that we maintain soundness by not removing any statement that might be relevant to security auditing when filtering rules are applied. Therefore, when defining the filtering rules below, we provide arguments on how we preserve *soundness*. Further, we empirically demonstrate the soundness in Section 3.4.

The five filtering mechanisms used in *JoanAudit* are:

F1 *(Irrelevant)* filters functions (custom functions or library APIs) that are irrelevant to the security analysis of XSS, SQLi, XMLi, XPathi, and LDAPi. Let M_{IR} be the set of irrelevant functions. During the SDG construction, upon encountering a node that corresponds to a function $f \in M_{IR}$, a stub node is generated instead of the PDG that represents f. By doing so, all the nodes and edges that correspond to f are filtered while not affecting the construction of the SDG. For security auditing purposes, the stub node is annotated with the name of the function and labeled as *irrelevant*.

F2 *(Known-good)* filters functions with known-good security properties. Let M_{KG} be the set of known-good functions. During the SDG construction, upon encountering a node that corresponds to a function $f \in M_{KG}$, a stub node is generated instead of the PDG that represents f. Therefore, like the filter above, all the nodes and edges that correspond to f are filtered in such a way as not to affect the construction of SDG. For security auditing purposes, the stub node is annotated with the name of the function and labeled as *known-good*.

Basically, the above two filters correspond to 1) functions that are known to be irrelevant to the auditing of XSS, SQLi, XMLi, XPathi, and LDAPi issues; and 2) functions that may be relevant to security but are known (or assumed) to be correct or free from security issues. Hence, it is clear that filtering such functions does not affect the soundness of our approach.

For example, we observed that Java methods belonging to classes responsible for retrieving the HTTP GET and POST parameters (e.g., those implementing the `javax.servlet.ServletRequest` interface) are commonly present in the original program chops; however — differently from the parameters they retrieve — these methods are irrelevant for our security analysis purpose because they contain neither input sanitization operations

nor security-sensitive operations concerning XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. Example functions excluded by the *known-good* filter are the ones provided by widely-used security libraries, such as Apache [8] and OWASP [94] (e.g., the methods of the classes implementing the `org.owasp.esapi.Encoder` interface); these functions are assumed to be correct and thus do not require auditing.

In our tool, we predefine 12 functions as *irrelevant* and 50 functions as *known-good*. Program developers or security auditors may need to extend these sets of functions based on their domain knowledge; these sets can be easily defined in our tool through a configuration file.

F3 (No input) filters sinks that are not influenced by any input source. This filtering is easily done by performing the program chopping with the source criterion I and the sink criterion k. The resulting chop $c(I, k)$ would be empty.

The sinks that are not influenced by any input sources cannot cause any security issues; thus, they are not relevant to security auditing. This implies that the resulting code, after applying *F3*, is still sound and yet more precise.

F4 (Declassification) filters out the secure paths from chop $c(I, k)$. Let $D_k \subseteq N$ be the set of declassifier nodes in SDG that corresponds to the type of sink k. Let P be a set of paths from input sources I to k. If there is a declassifier node $d \in D_k$ on a path $p \in P$, then the path p is removed from $c(I, k)$.

The presence of a declassifier on a path p in $c(I, k)$, which is adequate for securing the sink, ensures that values from input sources are properly validated and sanitized before being used in k, as far as path p is concerned. Hence, the resulting code after filtering such paths is still sound and yet more precise.

This filter is applied using the IFC analysis discussed in Section 3.2.2.1. We use information flow control to filter out— from the set of paths that are returned to the security auditor—the paths that do not contain any violation according to the $\mathcal{L}_{LH}$ lattice.

F5 (Automated fixing) It automatically fixes the paths from input sources I to sink k that can be identified as definitely vulnerable and that can be properly fixed without user intervention. Let P be the set of remaining paths from chop $c(I, k)$ after applying *F4*. If a path $p \in P$ identified as vulnerable can be fixed by applying an adequate security API, then the path p is removed from $c(I, k)$. This filter corresponds to the AUTOFIX procedure described in Section 3.2.2.2.

Automated fixing is not possible for all cases, especially when an input passes through complex string operations, like `substring` and `replace`, which are not addressed by our analysis. This is because there might be custom sanitization on the path using operations like `replace` and in that case, applying another sanitization procedure on the path could affect the integrity of the input data and may not fix the security issue as intended. Therefore, automated fixing is only applied for the inputs directly used in the sink or for the inputs that only pass through simple string operations like `concat`, `toUpperCase`, and `trim`, which do not have any (sanitization) effect on the input. For example, as discussed in Section 3.2.2.2, for the program in Figure 3.2, the fixing is applied to the input at Line 15 because it only passes through the `concat` and `toUpperCase` operations before it is used in

```
1  protected void doPost(
2      HttpServletRequest req,
3      HttpServletResponse res) {
4    PrinterWriter out = res.getWriter();
5    Document doc = DocumentBuilderFactory
6      .newInstance()
7      .newDocumentBuilder()
8      .parse("./students.xml");
9    XPath xpath=XPathFactory.newInstance()
10     .newXPath();
11   out.println("<html><head><title>"
12     + "Student␣Services" +
13     + "</title></head><body>...");
14   String op = req.getParameter("option");
15   String sid = req.getParameter("id");
16   int max = Integer.parseInt(req.
17     getParameter("max"));
18   String subj = req.getParameter("subjid");
19   if(!subj.substring(0,2).equals("cd"))
20       subj="*";
21   if(max>20)
22       max=20;
23   if(op.trim()
24      .equalsIgnoreCase("GradeQuery")) {
25     if(sid.length()>max) {
26         //remove chars <,>,/
27         sid=customSanit(sid);
28         out.println("<a␣href=\"foo.com?id="
29             + sid + "\">Invalid␣ID:␣"
30             + sid + "</a>"); //XSS sink
31         // ... other statements
32     } else if(sid.contains("id")){
33         sid=ESAPI.encoder()
34           .encodeForXPath(sid);
35         String query = "//students/grade[sid="
36             + sid + "␣and␣subjid='"
37             + subj +"']/mark";
38         NodeList nl=(NodeList)xpath
39           .evaluate(query, doc); //XPath sink
40         //... other statements
41     }
42   }
43 }
```

Listing 3.1: A Java servlet program with two sinks.

the sink. Fixing is also not possible when our analysis cannot determine the appropriate sanitization procedure to use, for example when it cannot identify the matching context due to complex code.

Anyway, since we apply the filter only on the paths that can be appropriately fixed, the resulting report after this filter is still sound and yet more precise for security auditing.

3.2.3 Application to an Example

In this section, we apply security slicing on the program illustrated in Listing 3.1 which contains the two sinks `out.println` at Line 30 and `xpath.evaluate` at Line 39.

Figure 3.4a and Figure 3.4b show the security slices for the sink at Line 30 and the sink at Line 39 in Listing 3.1, respectively. In both cases, the security slicing procedure filtered out library code from the `Document`, `HttpServletRequest`, `HttpServletResponse` and `PrinterWriter` classes (*F1*), since they can be considered as irrelevant to security.

Thanks to *F3*, two of the four program paths leading to the sink in Line 30 can be filtered out; the two pruned paths correspond to those including the predicate at Line 19 in Listing 3.1, which does not affect the sink (*F3*). As a result, the security slice shown in Figure 3.4a contains only two paths leading to the sink. For Figure 3.4b, *F3* pruned the block in Lines 27–30 since it does not affect the sink in Line 39.

This example shows that thanks to security slicing, large portions of the original program can be pruned away. Instead of auditing the whole program in Listing 3.1, the security auditor can focus her attention only on the security slices illustrated in Figure 3.4.

```
1  protected void doPost(
2    HttpServletRequest req,
3    HttpServletResponse res) {
4    PrinterWriter out = res.getWriter();
5
6
7
8
9
10
11   String op = req.getParameter("option");
12   String sid = req.getParameter("id");
13   int max = Integer.parseInt(req.
14     getParameter("max"));
15
16
17
18   if(max>20)
19       max=20;
20   if(op.trim()
21      .equalsIgnoreCase("GradeQuery")) {
22
23      if(sid.length()>max) {
24       //remove chars <,>,/
25       sid=customSanit(sid);
26       out.println("<a␣href=\"foo.com?id="
27         + sid + "\">Invalid␣ID:␣"
28         + sid + "</a>"); //XSS sink
29     }
30   }
31 }
32
33
34 //...
```

(a) Security slice of the XSS sink in Line 30.

```
1  protected void doPost(
2      HttpServletRequest req,
3      HttpServletResponse res) {
4
5    Document doc = DocumentBuilderFactory
6      .newInstance()
7      .newDocumentBuilder()
8      .parse("./students.xml");
9    XPath xpath=XPathFactory.newInstance()
10     .newXPath();
11   String op = req.getParameter("option");
12   String sid = req.getParameter("id");
13   int max = Integer.parseInt(req.
14     getParameter("max"));
15   String subj = req.getParameter("subjid");
16   if(!subj.substring(0,2).equals("cd"))
17       subj="*";
18   if(max>20)
19       max=20;
20   if(op.trim()
21      .equalsIgnoreCase("GradeQuery")) {
22     if(sid.length()<=max
23        && sid.contains("id")) {
24        sid=ESAPI.encoder()
25          .encodeForXPath(sid);
26        String query = "//students/grade[sid="
27            + sid + "␣and␣subjid='"
28            + subj +"']/mark";
29        NodeList nl=(NodeList)xpath
30          .evaluate(query, doc); //XPath sink
31     }
32   }
33 }
34 // ...
```

(b) Security slice of the XPath sink in Line 39.

Figure 3.4: Security slices for the sink at Line 30 (a) and Line 39 (b) sinks of the program in Listing 3.1.

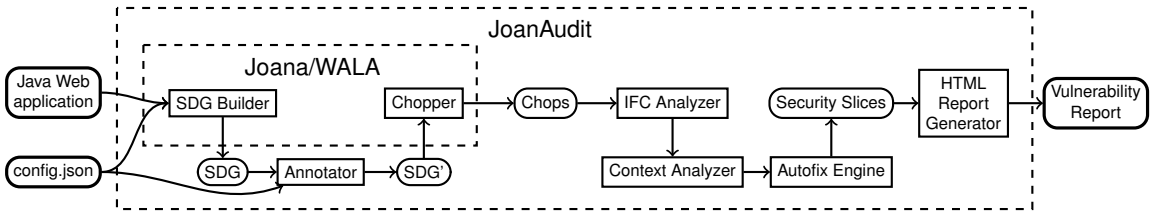


Figure 3.5: Architecture of *JoanAudit*.

3.3 Implementation

We have implemented our security slicing approach in a tool called *JoanAudit*; Figure 3.5 illustrates its architecture. The tool takes as input the bytecode of a Java Web application and a vulnerability catalogue (specified in the configuration file `config.json`), i.e., a pre-defined set of input source and sink signatures. For example, an input source could be the `getParameter` function from the Java Servlet API for accessing HTTP POST parameters; a possible sink could

be the `evaluate` function from the `javax.xml` package which executes XPath queries on an XML database.

JoanAudit first constructs an SDG in order to capture inter-procedural data-, control-, and call-dependencies. The SDG is derived from the Java bytecode by the SDG builder. The latter also prunes functions that are irrelevant to security (e.g., logging libraries) or functions that are known or assumed to be free from security issues (e.g., standard security libraries). The list of irrelevant and known-good functions is predefined but can be configured in the vulnerability catalogue.

Afterwards, the "Annotator" annotates the SDG with input sources, sinks, and declassifiers. Based on the annotations in the SDG, the tool generates a program chop for each sink. Each program chop contains all the program statements that influence a sink, starting from the input sources, possibly through different program paths. Sinks that are not affected by any input source are pruned from the SDG.

The block labeled "IFC Analyzer" performs IFC on each chop to determine if there are paths in the chop that can be considered secure due to the proper usage of sanitization functions and thus pruned. This step relies on a pre-defined set of declassifiers (standard sanitization procedures for preventing common injection vulnerabilities), which are configured in the vulnerability catalogue.

The block labeled "Context Analyzer" performs context analysis on the remaining paths. As part of this analysis, the block "Autofix Engine" attempts to patch, when feasible, the source code with the required security API. More specifically, this step uses *context analysis* to identify the context in which the data from an input source is used in the sink. Based on the identified context, this technique is able to automatically fix a vulnerable input source by applying the appropriate sanitization function to it. This technique is always guaranteed to properly fix a given vulnerability because it applies a fix only 1) in case of a direct data flow from an input source to a sink, and 2) if the context of the user input can be determined.

As output, the tool generates a report that guides the security auditor in auditing potentially vulnerable parts of the program. Figure 3.6 shows the main page of the report generated by *JoanAudit*, which gives an overview of all the paths from the security slices that were extracted by *JoanAudit*. The report indicates how many potentially vulnerable paths have been detected. Every row in the overview table represents a single path; it details the location of the sources and sinks in the source code, i.e., a combination of the scope or class in which the source/sink was found, and the line number of the source file. Moreover, the report indicates the path size (in terms of program statements) and vulnerability to which the path may be vulnerable.

After clicking on one of the rows in the overview table, the detailed information for the respective paths is displayed in an extra window (Figure 3.7), which shows the actual source code of the analyzed program and highlights the individual program statements belonging to the selected path. In this view, the source code line numbers are shown on the left; the scope, i.e., the class where the potential vulnerability has been found, is displayed at the top; source and sink, respectively, are the first and last highlighted statements in the code snippet. Notice that only the security-relevant parts are highlighted. This detailed view guides the security auditors from the input source to the potentially vulnerable sink.

The implementation of *JoanAudit* comprises approximately 11 kLOC (excluding library code) and is based on *Joana* [47, 43] and IBM's *Wala* framework [55]; *Joana* provides APIs for SDG generation from Java bytecode, program slicing, and IFC analysis.

id	Source	Sink	Len	Vulnerability
0	simple/Simple.java:96	simple/Simple.java:101	8	xss
1	simple/Simple.java:95	simple/Simple.java:117	18	xss
2	simple/Simple.java:96	simple/Simple.java:125	11	xpath injection
3	simple/Simple.java:95	simple/Simple.java:101	9	xss
4	simple/Simple.java:96	simple/Simple.java:117	17	xss
5	simple/Simple.java:95	simple/Simple.java:125	12	xpath injection
6	simple/Simple.java:95	simple/Simple.java:107	10	xss
7	simple/Simple.java:96	simple/Simple.java:116	16	xss
8	simple/Simple.java:96	simple/Simple.java:107	9	xss
9	simple/Simple.java:95	simple/Simple.java:116	17	xss

Figure 3.6: The overview page of the report generated by *JoanAudit* shows all potentially vulnerable paths found.

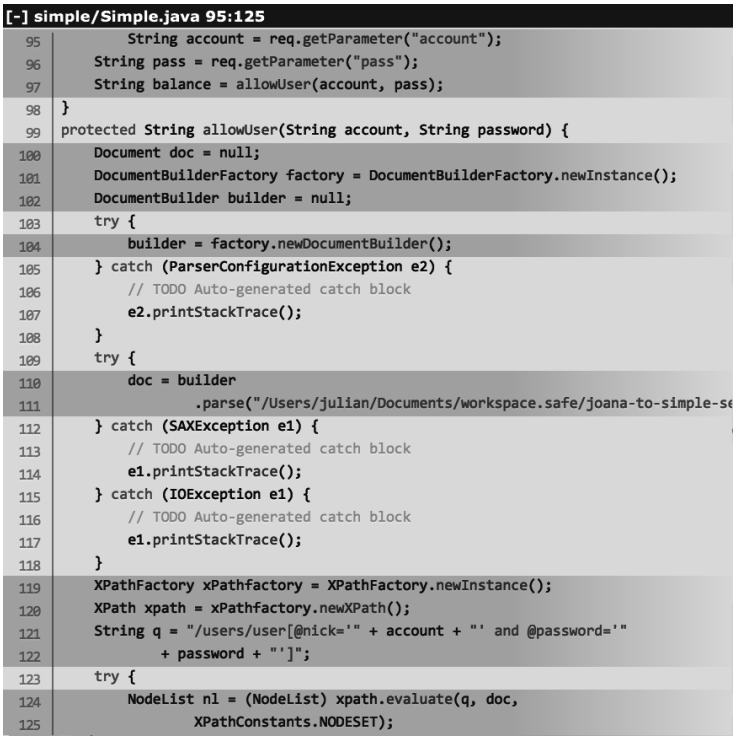


Figure 3.7: Security slice containing only the program statements relevant to security (highlighted statements).

The tool is configured with the JSON file `config.json` which contains a list of Java bytecode signatures for input sources, sinks, and declassifiers. The `config.json` file also specifies the list of bytecode signatures for known-good and irrelevant APIs. Note that *JoanAudit* is highly customizable: based on their domain knowledge, developers can specify in `config.json` additional input sources, sinks, and custom declassifiers used in their applications. Thanks to this user-defined additional configuration, the tool will not skip analyzing other security-sensitive operations, and will not falsely report as insecure the paths containing custom declassifiers. The excerpt shown in Listing 3.2 from the `config.json` file shows an example configuration for the sink corresponding to the function `evaluate` (from the `javax.xml` package).

```
1 "sinks":[{ "name":
2 "javax.xml.xpath.XPath.evaluate(Ljava/lang/String,Ljava/lang/Object;)Ljava/lang/String;",
3 "labels": "1(H)" }]
```

Listing 3.2: *JoanAudit* configuration file `config.json`.

The configuration entry for the sink is a JSON object with a `name` attribute, i.e., the bytecode signature of the sink (in bytecode format), and a `labels` attribute that specifies the index of the parameter to be tracked and its security level. The security level is important for IFC in order to detect an actual security violation. In the example configuration, we label the first parameter of the `evaluate` function with security label `H` (high integrity) which requires that data arriving at the sink should not be tampered with. The configuration for sources and declassifiers is done similarly; the detailed overview of the configuration is available at `https://github.com/julianthome/joanaudit`.

3.4 Evaluation

In this section, we present the evaluation of our security slicling approach: Section 3.4.1 defines the research questions; Section 3.4.2 presents the benchmark applications, and in Section 3.4.3 we show and discuss the evaluation results.

3.4.1 Research Questions

To evaluate whether our approach achieves precision, soundness and run-time performance when providing assistance to security auditing, we aim to answer the following research questions:

RQ1 *(Precision)* How much reduction can be expected from security slicing in terms of source code to be inspected? Is the reduction practically significant?

RQ2 *(Soundness)* Do we extract all the statements that are relevant to auditing XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities?

RQ3 *(Performance)* Does the tool scale to realistic systems in terms of run-time performance?

3.4.2 Test Subjects

Table 3.2 shows the 9 Web applications/services that we used in our evaluation. *WebGoat* [96] is a deliberately in-secured Web application/service for the purpose of teaching security vulnerabilities. It contains various realistic vulnerabilities that are commonly found in Java Web applications. Apache *Roller* [7] and *Pebble* [97] are blogging applications that also expose a Web service APIs. *Regain* [104] is a search engine that allows users to search for files over a Web front-end. *PSH* [100] is the implementation of the open protocol *PubSubHubbub* for distributed publish/subscribe communication [45], which is supported by many blogging applications and

Table 3.2: Test subjects.

	Java LOC	#Prog.	#Sources	#Sinks						#Declassifiers					
				XML	XPath	XSS	LDAP	SQL	others	XML	XPath	XSS	LDAP	SQL	others
WebGoat 5.2	24,608	14	34	1	1	35	0	29	2	0	0	0	0	21	0
Roller 5.1.1	52,433	3	14	10	0	13	0	0	0	8	0	3	0	0	0
Pebble 2.6.4	36,592	3	6	0	0	6	0	0	1	0	0	0	0	0	3
Regain 2.1.0	23,182	1	3	0	0	1	0	0	0	1	0	2	0	0	0
PSH 0.3	1,964	3	3	10	2	0	0	0	0	2	0	0	0	0	0
TPC-App	2,082	6	22	0	0	2	0	7	0	0	0	0	0	11	0
TPC-C	9,184	6	16	0	0	0	0	24	0	0	0	0	0	58	0
TPC-W	2,470	6	6	0	0	0	0	6	0	0	0	0	0	6	0
RAP	442	1	2	0	0	0	4	0	0	0	0	0	0	0	0
Total	152,957	43	106	21	3	57	4	66	3	11	0	5	0	96	3

also used to access newsfeeds on the Internet. *RAP* [110] is an LDAP-based Web service that authenticates users against an LDAP directory.

We selected *WebGoat*, Apache *Roller*, and *Pebble* since they are commonly used as benchmarks for security [78, 138, 77, 142, 137, 85]. The choice of *Regain* was driven by the fact that it is used in a production-grade system by *dm*, one of the biggest drugstore chains in Europe. *TPC-App*, *TPC-C*, and *TPC-W* are the benchmarks used by Antunes and Vieira [6] for evaluating vulnerability detection tools for Web services; these benchmarks contain a set of Web services accepted as representative of real environments by the Transactions processing Performance Council (`http://www.tpc.org`). The *PSH* tool was chosen because it is the most popular Java project related to the *PubSubHubbub* protocol in the Google Code archive [42]. Similarly, we selected *RAP* because it was one of the first Java projects returned by a query on Github.com with the search string `ldap rest`.

Table 3.2 also reports the sizes of the test subjects in terms of LOC, excluding the library code. The test subjects have an average size of 17 kLOC, and the largest one has 52 kLOC, which is fairly typical for that type of systems. The third column in Table 3.2 shows the numbers of Web programs (#*Prog.*), i.e., Java Server Pages (JSP), Java servlets and classes, contained in each test subject and analyzed by our tool *JoanAudit*. The table also reports the numbers of input sources (#*Sources*), sinks (#*Sinks*), and declassifiers (#*Declassifiers*) that *JoanAudit* identified. For sinks and declassifiers, the numbers are shown separately with respect to XSS, SQLi, XMLi, XPathi, and LDAPi. Some sinks are very general and are exploitable in various ways (e.g., sinks that allow attackers to load arbitrary classes server-side). Due to their universality, we also considered them in our evaluation and their number is listed in column *others* in Table 3.2.

All these test subjects can be obtained from the tool website [127].

3.4.3 Results

We ran our evaluation on a Apple MacBook Pro with an Intel Core i7 (2 GHz) and 8 GB of RAM, running Mac OS X 10.11, JVM version 25.31-b07, *Joana* rev. 688, *Wala* v.1.1.3, and OWASP ESAPI 2.0.

3.4.3.1 Precision

To answer *RQ1*, we compared the size of the slices produced by *JoanAudit* (hereafter referred to as "security slices") with the size of the slices produced by the state-of-the-art chopping implementation provided by *Joana* (hereafter referred to as "normal chops") extended with source/sink identification capabilities; in terms of size, we considered both the number of nodes and the number of edges. More specifically, for each sink k, we computed a security slice using our approach and a normal chop with the criterion (I, k). We used the *Wilcoxon signed-rank test* over the slice sizes across Web programs in order to determine whether the differences in sizes of the two types of slices were statistically significant. We also discuss whether this difference is of practical significance in terms of auditing effort.

As shown in Table 3.2, we analyzed 43 Web programs from the 9 test subjects. For each Web program, an SDG was constructed. We computed normal chops and security slices from each SDG. The results are shown in Table 3.3. Overall, we computed 154 normal chops (#*ch*) and 39 security slices (#*ss*) from 106 sources and 154 sinks. The size (in terms of #nodes and #edges) of SDGs, normal chops, and security slices are shown in columns *SDG*, *Chopping*, and *SecuritySlicing*, respectively. Column #*ss* reports the final output of *JoanAudit*, i.e., the numbers of remaining security slices that require auditing after filtering has been performed. Some of the computed security slices are completely filtered (i.e., #ss=0) when, for example, all the paths in a slice are detected to be secured because of the presence of declassifiers. Furthermore, the last four columns in Table 3.3 show the effectiveness of the five different filters presented in Section 3.2.2.3, in terms of the number of nodes that are filtered.

To determine the amount of reduction achieved by security slicing when compared to normal chopping, we computed the relative size reduction of security slices with respect to (unfiltered) normal chop. The results (in percentage) are given in the columns *(N%)* and *(E%)* in Table 3.3. These results show that our security slices are significantly smaller than their counterparts obtained through normal chopping, in terms of both the number of nodes and the number of edges. As shown in the last two rows of the table, our approach achieved mean and median reductions of $76\,\%$ and $100\,\%$, respectively, in terms of the number of nodes, and $79\,\%$ and $100\,\%$, respectively, in terms of the number of edges. Note that for 25 out of 43 cases, no security slices were reported which explains the median reduction of 100%. A reduction of 100% is possible for those cases where security slicing automatically prunes all paths which can be considered to be properly sanitized (*F4*) and/or cases where the automated fixing filter is applicable (*F5*).

115 chops were completely dropped by the filters, meaning that only 39 out of total 154 chops require manual auditing (see columns #*ch* and #*ss*). Hence, one can expect significant practical benefits by adopting our approach. The Wilcoxon signed-rank tests over 43 observations (#*Prog*.) show that the size reductions achieved with security slices are statistically significant at a 99% level of significance.

From the last four columns in Table 3.3, we can also observe how much each type of filters contributed. The *known-good* and *irrelevant* library-code-filters (*F1+F2*) significantly reduced the SDG size for all the test subjects. This can be explained by the fact that applications typically contain a large chunk of library code. The *no input* filter (*F3*) also significantly pruned many nodes ($74\,776$ nodes in total) since those nodes are not influenced by any input source. The *declassification* filter (*F4*) significantly pruned many nodes from the standard chops (3645 nodes

```
1 public AuthResponse authenticatePost(
2   @FormParam("user") String user, // SOURCE
3   @FormParam("pass") String pass
4 ) {
5   // ...
6   LdapAuthentication ldap = getLdap(user);
7   // ...
8   ldap.authenticate(user,pass);
9   // ...
10 }
11
12 private LdapAuthentication getLdap(String user) {
13   // ...
14   String sfilter = Configuration.get(Keys.LDAP_SFILTER);
15   LdapAuthentication ldap = new LdapAuthentication();
16   // ...
17   if (!StringUtils.isEmpty(sfilter))
18     ldap.setSearchFilter(sfilter.replaceAll("{user}", user)); //SINK
19   // ...
20   return ldap;
21 }
```

Listing 3.3: Security slice from the *RAP* `/LdapAuthService`.

in total), for all the test subjects except *RAP* . The *automated fixing* filter (*F5*) was significant for *WebGoat*, *PSH*, and *TPC-W* (751 nodes were pruned in total).

To conclude, by comparing the security slice sizes and the SDG sizes in Table 3.3, we can observe that on average security slicing would require the audit of approximately 1% of the code for all the sinks in a given Web application. Since the security slices computed by our approach are based on the control-flow paths between sinks and sources, the size reduction of security slicing achieved with *JoanAudit* is directly correlated to the reduction of the manual effort required from security auditors for verifying vulnerable paths in the source code. Hence, these results answer *RQ1* by clearly suggesting that a significant reduction in code inspection can be expected when using our approach.

We also remark that the above comparison shows the benefit of security slicing over normal chopping, with the latter performed by using a tool (*Joana*) that is also not easy to configure and use for standard engineers. Furthermore, for situations where security auditors have no access to program chopping tools, our approach can also indicate the percentage of the entire program code that has to be audited with security slices.

3.4.3.2 Soundness

To answer *RQ2*, we manually inspected all the security slices (39) returned by *JoanAudit* and compared them to their normal chop counterparts, to determine whether our security slicing approach had pruned any information relevant to auditing XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. To illustrate this manual inspection process, we use the simplified code excerpt shown in Listing 3.3, which corresponds to a security slice extracted from the *RAP* / `LdapAuthService` program by *JoanAudit*.

In the code above, function `authenticatePost` can be called by a user to request authentication with the *RAP* web service; its inputs are the username (`user`, Line 2) and the password (`pass`, Line 3). Function `getLdap` creates an `LdapAuthentication` object, which manages all

Table 3.3: Comparison between the size of the slices obtained with normal chopping and the size of the slices obtained with security slicing (#ch: number of normal chops; N%: reduction of nodes in percentage; E%: reduction of edges in percentage; #ss: number of security slices; F1–F5: numbers of nodes filtered by each of the proposed five filters).

	SDG		Chopping			SecuritySlicing					Filtering			
Program Name	Nodes	Edges	Nodes	Edges	#ch	Nodes	(N%)	Edges	(E%)	#ss	F1+F2	F3	F4	F5
WebGoat	160,573	923,709	16,359	19,405	68	3,902	76	3,916	80	21	133,389	21,007	1,746	529
1.BackDoors	11,196	63,350	210	229	1	171	19	172	25	1	10,367	658	0	0
2.BlindNumericSqlInjection	9,573	52,262	721	813	6	0	100	0	100	0	7,637	1,600	211	125
3.BlindScript	21,558	140,134	1,072	1,296	3	318	70	322	75	3	20,634	606	0	0
4.BlindStringSqlInjection	9,616	52,580	721	813	6	0	100	0	100	0	7,654	1,626	211	125
5.InsecureLogin	11,998	68,257	2,205	2,630	5	673	69	673	74	2	9,864	1,410	51	0
6.MultiLevelLogin1	13,525	80,281	969	1,341	4	0	100	0	100	0	11,918	1,126	481	0
7.MultiLevelLogin2	12,546	71,773	1,696	2,172	6	670	60	676	69	1	9,263	2,504	109	0
8.SqlAddData	10,565	58,219	1,535	1,756	8	169	89	170	90	2	8,617	1,365	336	78
9.SqlModifyData	10,623	58,350	1,606	1,827	12	233	85	234	87	3	8,549	1,386	343	112
10.SqlNumericInjection	13,576	77,717	1,712	2,028	5	376	78	376	81	2	11,845	1,354	1	0
11.SqlStringInjection	12,155	69,502	2,134	2,479	5	567	73	567	77	3	9,923	1,664	1	0
12.WsSAXInjection	8,075	45,164	833	940	3	352	58	352	63	2	4,448	3,274	1	0
13.WsSqlInjection	9,191	49,232	820	940	3	373	55	374	60	2	7,338	1,479	1	0
14.XPATHInjection	6,376	36,888	125	141	1	0	100	0	100	0	5,332	955	0	89
Roller	16,361	142,811	2,562	3,110	23	353	86	353	89	1	12,614	2,812	582	0
15.CommentDataServlet	11,119	115,398	1,354	1,607	12	353	74	353	78	1	9,242	1,298	226	0
16.AuthorizationServlet	752	3,578	101	120	1	0	100	0	100	0	97	651	4	0
17.OpenSearchServlet	4,490	23,835	1,107	1,383	10	0	100	0	100	0	3,275	863	352	0
Pebble	1,605	7,824	560	717	7	3	99	2	100	1	529	986	87	0
18.ImageCaptchaServlet	829	4,033	536	697	1	0	100	0	100	0	470	293	66	0
19.SecurityUtils	236	1,128	21	18	5	0	100	0	100	0	28	187	21	0
20.XmlRpcController	540	2,663	3	2	1	3	0	2	0	1	31	506	0	0
Regain	43,197	622,748	474	568	1	0	100	0	100	0	28,562	14,458	177	0
21.FileServlet	43,197	622,748	474	568	1	0	100	0	100	0	28,562	14,458	177	0
PubSubHubbub	3,313	17,281	207	208	12	0	100	0	100	0	2,209	899	142	63
22.Discovery	160	726	63	63	2	0	100	0	100	0	0	97	0	63
23.Publisher	1,896	10,097	45	44	5	0	100	0	100	0	1,405	446	45	0
24.Subscriber	1,257	6,458	99	101	5	0	100	0	100	0	804	356	97	0
TPC-App	190,177	1,198,618	1,125	1,309	9	99	91	97	93	2	161,378	28,459	198	43
25.ChangePaymentMethod_Vx0	9,671	56,074	166	179	2	0	100	0	100	0	9,368	165	138	0
26.ChangePaymentMethod_VxA	10,151	58,890	49	48	1	49	0	48	0	1	9,773	329	0	0
27.ProductDetails_Vx0	10,330	59,197	420	506	2	0	100	0	100	0	10,103	183	44	0
28.ProductDetails_VxA	10,554	60,414	434	522	2	50	88	49	91	1	10,316	185	3	0
29.NewProducts_Vx0	74,609	481,203	13	12	1	0	100	0	100	0	60,803	13,793	13	0
30.NewProducts_VxA	74,862	482,840	43	42	1	0	100	0	100	0	61,015	13,804	0	43
TPC-C	92,559	568,680	1,860	1,932	24	1,044	44	1,048	46	10	87,424	3,471	620	0
31.Delivery_Vx0	13,606	81,511	266	276	7	0	100	0	100	0	12,577	775	254	0
32.Delivery_VxA	16,130	97,431	493	503	3	405	18	408	19	3	14,903	822	0	0
33.OrderStatus_Vx0	18,963	120,016	287	301	5	0	100	0	100	0	18,083	614	266	0
34.OrderStatus_VxA	20,395	129,702	476	490	5	455	4	457	7	5	19,287	653	0	0
35.NewStockLevel_Vx0	11,266	67,071	127	139	2	0	100	0	100	0	10,871	295	100	0
36.NewStockLevel_VxA	12,199	72,949	211	223	2	184	13	183	18	2	11,703	312	0	0
TPC-W	63,290	365,728	213	209	6	0	100	0	100	0	60,698	2,383	93	116
37.DoSubjectSearch_Vx0	10,347	59,748	26	25	1	0	100	0	100	0	9,947	374	26	0
38.DoSubjectSearch_VxA	10,549	60,854	40	39	1	0	100	0	100	0	10,132	377	0	40
39.DoAuthorSearch_Vx0	10,541	60,790	49	50	1	0	100	0	100	0	10,118	378	45	0
40.DoAuthorSearch_VxA	10,549	60,854	40	39	1	0	100	0	100	0	10,132	377	0	40
41.GetCustomer_Vx0	10,551	61,187	22	21	1	0	100	0	100	0	10,092	437	22	0
42.GetCustomer_VxA	10,753	62,295	36	35	1	0	100	0	100	0	10,277	440	0	36
RAP	655	2,838	354	378	4	332	6	343	9	4	22	301	0	0
43.LdapAuthService	655	2,838	354	378	4	332	6	343	9	4	22	301	0	0
Total	571,730	3,850,237	23,714	27,836	154	5,773		5,759		39	486,825	74,776	3,645	751
Mean	13,296	89,540	551	647	4	133	76	134	79	1	11,322	1,739	85	17
Median	10,551	60,790	287	301	3	0	100	0	100	0	9,923	651	21	0

the communications with the LDAP backend server and stores configuration attributes that are important for user authentication (e.g., distinguished name, search filter, LDAP host address, port). First, the pre-configured search filter is loaded from the configuration file (Line 14); then,

Table 3.4: Execution time of the individual steps in *JoanAudit* (in ms).

	SDG Generation	Source/Sink Identification	Chopping	Filtering	Total
WebGoat	21,774	201	59,427	42,278	123,680
Roller	5,079	64	16,125	1,241	22,509
Pebble	2,949	21	234	40	3,244
Regain	4,315	20	758	354	5,447
PSH	2,876	41	367	224	3,508
TPC-App	16,297	112	2,157	4,349	22,915
TPC-C	8,089	63	3,931	6,664	18,747
TPC-W	7,590	31	313	3,044	10,978
RAP	945	6	6,220	25,765	32,936
Mean	7,768	62	9,948	9,329	27,107

an `LdapAuthentication` object is created (Line 15). The pre-configured search filter can contain placeholders surrounded by curly brackets that are replaced with concrete values. For example, given the search filter `(&(objectClass=inetOrgPerson)(uid={user}))`, the placeholder `{user}` is replaced with the value provided with parameter `user` at Line 18, and then the result is stored in the `LdapAuthentication` object through the `setSearchFilter` method.

We started our manual inspection process at the sink (Line 18), to determine the variables it uses (`sfilter` in the example above). Then, we tracked back its dependent statements to identify how the variables were processed. We determined that there was an unsanitized input at Line 2 on which the sink in Line 18 is data dependent. Hence, a user could alter the semantics of the search filter `sfilter` by injecting LDAP filter fragments such as `( ) * & |` through the `user` variable at Line 2. There was no known LDAPi vulnerability reported before for *RAP* ; by using our tool, we detected a new LDAPi vulnerability and reported it to the developers.

In addition to inspecting security slices, we also manually inspected all the normal chops (154 chops) to determine if our security slicing had incorrectly dropped the whole chop from being reported (i.e., generating a false negative). Following a similar process, we verified that our security slicing approach neither missed any information important for security auditing nor incorrectly dropped any chop: this answers *RQ2*.

3.4.3.3 Performance

To answer *RQ3*, we measured the time taken for performing each step in the generation of security slices and normal chops; the results are shown in Table 3.4. *JoanAudit* took an average of 27 s to analyze individual test subjects and required a maximum of 124 s to analyze the largest one. These results show that *JoanAudit* exhibits good run-time performance, which makes it suitable to analyze Java Web applications similar in size to our test subjects, which is the case for many such systems.

Furthermore, we remark that the sum of the values in the columns "SDG Generation", "Source/Sink Identification", and "Chopping" corresponds to the execution time of the state-of-the-art chopping implementation provided by *Joana* extended with source/sink identification capabilities (i.e., *normal chopping*). The difference between this approach and ours lies only in the extra time taken by the filtering step, which on average accounts for 33% of the total time.

3.4.3.4 Threats to Validity

Our empirical evaluation is subject to threats to validity. The results were obtained from 9 selected Web applications, and hence, they cannot necessarily be generalized to all Web applications. We minimized this threat by choosing test subjects that vary in sizes and functionalities, and by picking realistic Java projects, which in many cases represent well-known benchmarks in the context of security.

We compared our approach, in terms of size reduction and performance, with a state-of-the-art chopping implementation provided by *Joana* extended with source/sink identification capabilities. Note that we expect, however, to achieve similar results when comparing with other Java program slicing/chopping tools (e.g., *Indus* [59]) since our approach works on top of program chopping and is independent from the specific chopping tool we use.

Lastly, since our security slicing approach and tool are targeted towards Java Web applications, the approach may not produce the same results for Web applications written in other languages. Nevertheless, the fundamental principles of our approach are not language-specific and can be adapted to other languages using the corresponding program slicing tools (e.g., *CodeSurfer* [126] for C++).

3.5 Summary

In this chapter, we presented security slicing to assist the security auditing of common injection vulnerabilities, namely XSS, SQLi, XMLi, XPathi, and LDAPi. For every security-sensitive operation in the program, we extract a sound and precise slice, along with path conditions, to help analysts perform security auditing on minimal chunks of source code. This is meant to be complementary to current vulnerability detection approaches by helping the auditor identify false positives and negatives. A prototype tool that automates our approach was fully implemented and was used to generate 39 security slices from 43 Web programs. In comparison with conventional program slices, we observed that our security slices are 76 % smaller on average while still retaining all the information relevant for verifying common vulnerabilities. We also made the tool and the test subjects available online so that researchers can validate and build on our results.

Chapter 4

Search-driven String Constraint Solving

The chapter introduces search-driven constraint solving and is organized as follows:

Section 4.1 gives an overview of our string constraint solving algorithm. Section 4.2 discusses the motivations for this work and provides an example. Section 4.3 illustrates our search-driven approach for string constraints solving. Section 4.4 illustrates the implementation of *ACO-Solver*, i.e., the implementation of our search-driven constraint solving approach; Section 4.5 presents the evaluation of our approach; Section 4.6 concludes this chapter.

4.1 Overview

State-of-the-art approaches [64, 112, 40, 153] for identifying security vulnerability are based on symbolic execution and constraint solving. Roughly speaking, these approaches consist of solving the constraints corresponding to the attack condition, obtained by conjoining the path conditions generated by the symbolic execution with an approprate threat model (see Section 2.1.3). In case the solver yields SAT, showing the satisfiability of the attack condition, it means that the attack is feasible and that the analyzed path is vulnerable to the attack. The main strength of this approach is that vulnerability detection yields a limited number of false positives, since the concrete inputs determined with constraint solving prove the existence of vulnerabilities.

However, the effectiveness and precision of these approaches are challenged by the *degree of support for (complex) string operations provided by the constraint solver itself*. State-of-the-art solvers such as Kaluza [112], Stranger [149], *CVC4* [74], S3 [136], and *Z3-str2* [152] support only a limited number of strings operations, such as concatenation, assignment, and equality; more complex operations like string replacement or standard sanitization functions are not supported or only partially-supported. Existing solvers could be extended to provide native support for complex string operations, but the task is non-trivial and not scalable to the size of a complete string function library of a modern programming language, or of sanitization libraries like OWASP ESAPI [94] and Apache Commons Lang [8]; for example, the classes `String`, `StringBuffer`,

`StringBuilder` from the Java Standard Library, and the classes `StringUtils`, `StringEscapeUtils` from the Apache Commons Lang library contain a total of 370 methods.

Alternatively, complex string operations could be transformed into a set of equivalent constraints with only operations natively-supported by the solver; however, such a solution would increase the complexity of the generated constraints, potentially leading to scalability issues [40]. In practice, existing solvers fail (i.e., they crash or return an error) when they encounter an unsupported operation; in the context of vulnerability detection, this behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

The goal of the approach presented in this chapter is to address the challenge of supporting complex operations in string constraint solvers by proposing a search-driven constraint solving technique that *complements* them. We intentionally target a solution that does not rely on any assumption regarding the selected constraint solver and that can therefore be widely used in the future.

The idea is to solve the constraints (in an attack condition) through a two-stage process. In the first stage we take any constraint solver and use it to solve the constraints that contain only operations supported by the solver itself. The remaining unsolved constraints, which contain operations not supported by the solver, are handled in the second stage, by means of a *hybrid* constraint solving procedure.

We treat the solver in the first stage as a black-box, and only assume that it terminates its execution either by failing (when it encounters a constraint containing an operation that it does not support) or by returning an answer (which can be either UNSAT or SAT and a solution). The hybrid solving procedure in the second stage is executed only when the solver in the first stage fails. In the second stage we solve the constraints containing unsupported operations by means of a *hybrid* search-driven procedure that leverages the ACO meta-heuristic [36]. This procedure searches for a solution that satisfies the constraints involving unsupported operations; the search is driven by different fitness functions, depending on the type of the constraints. We call this procedure *hybrid* because we reduce its search space before running the search itself, to make the latter scalable. We perform the search space reduction by restricting the domains of the string variables involved in the constraints to solve. To do so, in our current strategy and given the state of the art, we rely on an automata-based string constraint solver (*Sushi* [40]).

4.2 Motivating Example

In this section we present a motivating example that highlights the need to handle complex string constraints in the context of vulnerability detection based on constraint solving. Although we crafted this example for illustrative purposes, it can be considered realistic since it contains typical operations that are commonly found in modern Web applications.

The program, shown in Listing 3.1, contains two sinks. The first sink is at Line 30 and corresponds to an XSS vulnerability within an HTML output operation; the second one is at Line 39 and corresponds to an XPathi vulnerability within an XPath query.

The sink at Line 39 is vulnerable to XPathi because the variable `sid`, containing a user input, is not sanitized properly before using it in the XPath query. Indeed, the standard sanitization procedure `ESAPI.encoder().encodeForXPath` from OWASP [94] applied to variable `sid` only escapes meta-characters such as `'` and `"`. Assuming that the element `sid` is defined as a numeric data type in the schema of the document `students.xml` (the same presented in Section 2.1.1),

one could still perform a successful attack without using those meta-characters, for example using the input `1 or 0<1`. This example shows that sanitization, even when achieved by applying widely used and well-tested sanitization libraries, does not always work. Indeed, sanitization libraries often provide operations that filter user input *only* based on a certain context (an XPath attribute in the example above), without necessarily considering all possible cases.

The XPathi vulnerability in the example can be discovered by using symbolic execution and constraint solving, combined in a three-step procedure:

1. *Path conditions generation through symbolic execution*. For example, one of the path conditions generated by symbolically

   executing a path condition leading to the execution of the XPath sink at Line 39 is:

   $$\begin{aligned} PC0 \equiv\ & SUBJ\texttt{.substring(0,2).equals("cd")} \wedge \\ & \texttt{Integer.parseInt(}MAX\texttt{)} = 20 \wedge \\ & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \wedge \\ & SID\texttt{.length()} \leq 20 \wedge \\ & SID\texttt{.contains("id")} \end{aligned}$$

   where $OP, SID, MAX, SUBJ$ are symbolic values for the variables initialized with the Web request inputs (lines Lines 14–18).

2. *Definition of the attack condition*. In this step, usually performed by a security expert[1], the attack condition is defined in a way that properly characterizes security threats. It can be done by writing attacks that match any of the attack patterns described in Section 2.1.3.

   For $PC0$ the security attack `matches(".* or 1=1 .*")` describes a tautology attack pattern (an instance of threat model #20 in Table 2.1). The resulting attack condition can be described as follows:

   $$\begin{aligned} ATTK0 \equiv & \texttt{ESAPI.encoder().encodeForXPath(}SID\texttt{)} \\ & \texttt{.matches(".* or 1=1 .*")} \end{aligned}$$

   where `ESAPI.encodeForXPath(`SID`)` is the symbolic expression over the symbolic value SID representing the values of variable sid at the sink.

3. *Constraint solving*. The third step requires to solve the *attack condition*, defined as the constraint obtained by conjoining the path condition with the attack specification; this step is performed using a constraint solver. If the solver yields SAT, showing the satisfiability of the constraint, it means that the attack is feasible and that the analyzed path is vulnerable to the attack. In the example, the constraint $SEC0 \equiv PC0 \wedge ATTK0$ is satisfiable, confirming the presence of XSS and XPath vulnerabilities, respectively.

This procedure assumes that the constraint solver is able to handle string operations like `trim`, `toLowerCase`, `parseInt`, `equalsIgnoreCase`, `length` and `encodeForXPath`. However,

[1]This step needs to be done once for each type of vulnerability, and possibly refined over time if needed.

state-of-the-art solvers such as Kaluza [112], CVC4 [74], S3 [136], and *Z3-str2* [152] do not support at least one of these complex operations. From a more general standpoint, the major challenge faced when adopting a vulnerability detection procedure based on constraint solving is the *degree of support for (complex) string operations provided by the constraint solver itself*.

One way to face this challenge is to modify or enhance an existing solver in order to provide native support for complex string operations. However, this task is non-trivial and requires a deep understanding of string manipulating functions and constraint solving; moreover, it is not scalable to the size of a sanitization library like OWASP ESAPI or of a complete string function library of a modern programming language. Alternatively, instead of modifying the solver, one could re-express complex operations with their equivalent set of basic constraints that can be solved by the solver. Although relatively easier, this alternative still requires significant effort and expertise, and usually results in complex constraints that may still lead to scalability issues for constraint solvers [40]. For example, consider one of the constraints in the above path condition: OP`.trim().equalsIgnoreCase("GradeQuery")`; assuming the solver handles only `length`, `charAt`, `equals`, and `substring`, one could re-express this constraint as:

$$
\begin{aligned}
&\exists c_1, c_2, 0 \leq c_1 \leq c_2 \leq OP\texttt{.length()}, \text{ such that} \\
&(OP\texttt{.substring(}c_1\texttt{,}c_2\texttt{).equals("gradequery")} \\
&\vee \cdots \vee \ldots \texttt{equals("gRadeQueRy")} \ldots \vee \cdots \vee \\
&OP\texttt{.substring(}c_1\texttt{,}c_2\texttt{).equals("GRADEQUERY")}) \\
&\wedge \forall i, 0 \leq i < c_1, OP\texttt{.charAt(}i\texttt{)} = \texttt{' '} \\
&\wedge \forall j, c_2 < j \leq OP\texttt{.length()}, OP\texttt{.charAt(}j\texttt{)} = \texttt{' '}
\end{aligned}
$$

which uses equivalent constraints for `equalsIgnoreCase` and `trim`. Notice how the operation `equalsIgnoreCase` is expanded into a disjunction of constraints with the `equals` operation, which cover all the possible combinations of the characters denoting a case-insensitive representation of the string "GradeQuery"; also, modeling the `trim` operation requires to add several auxiliary variables and predicates.

To work around this issue, the current solution in practice is to have the constraint solver fail (i.e., it crashes or returns an error) when it encounters an unsupported operation. Our experiments show that this is the case for state-of-the-art solvers like *CVC4* and *Z3-str2*. However, in the context of vulnerability detection, such a behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

Hence, the challenge discussed above shows that, in the context of vulnerability detection, there is a need for scalable and precise techniques for constraint solving that can handle constraints with complex string operations.

4.3 Approach

We address the challenge of supporting complex operations in string constraint solvers — in the context of vulnerability detection — by proposing a search-driven constraint solving technique that *complements* their support for complex string operations.

The idea, illustrated in Figure 4.1, is to solve the constraints corresponding to an attack condition AC through a two-stage process. In the first stage we take any existing constraint solver

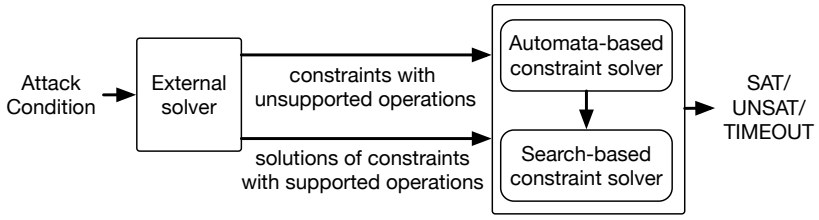


Figure 4.1: Two-stage approach for string constraint solving.

```
1: function CstrSolve(AttackCondition AC)
2:     Set of Solution Sol ← ∅
3:     Boolean externalSolved ← false
4:     Set of Set of Constraint H ← getDependentSetsOfCstrs(AC)
5:     for all H_i ∈ H do
6:         ⟨externalSolved, Sol⟩ ← externalSolve(H_i)
7:         if ¬externalSolved then
8:             ⟨H'_i, Sol⟩ ← automataSolve(H_i)
9:             if H'_i ≠ ∅ then
10:                Sol ← searchSolve(H'_i, Sol)
11:                if Sol = ∅ then
12:                    return TIMEOUT
13:                end if
14:            end if
15:        end if
16:    end for
17:    return ⟨SAT, Sol⟩
18: end function
```

Algorithm 4.4: Search-based constraint solving algorithm.

and use it to solve the conjuncts in AC that contain only operations supported by the solver itself. The remaining conjuncts in AC, which contain unsupported operations, are solved in the second stage, by means of a *hybrid* constraint solving procedure that combines an automata-based solver with a search-driven solving procedure based on the ACO meta-heuristic.

The meta-heuristic search in the second stage tries to find solutions for the variables involved in conjuncts of AC that contain operations that neither the solver in the first stage nor the automata-based solver in the first step of the second stage supports. Nevertheless, we invoke the automata-based solver *before* the meta-heuristic search in order to *reduce* the search space of the latter. We specifically use an automata-based (vs. bit-vector-based or word-based) constraint solver because it returns, when successful, a solution automaton for each variable occurring in the constraints it could solve, based on the operations it supports. This automaton accepts the language corresponding to the set of values (for the variable) that satisfy the constraints involving the operations that the solver supports. In this way, we are able to reduce (possibly in a significant way) the size of the domains of the variables involved in the constraint; this is expected to make the search more scalable and effective.

The search-based constraint solving algorithm is illustrated in Section 4.3.1, and Section 4.3.2 explains how ACO is applied for solving constraints that involve unsupported operations.

4.3.1 Algorithm

The pseudocode of our string constraint solving algorithm is shown in Algorithm 4.4. It takes in input an attack condition AC (expressed as a conjunction of constraints) and returns whether it is satisfiable, unsatisfiable, or whether it timed out; when it returns satisfiable, it also returns the set of solutions found.

First, it decomposes (Line 4) the attack condition AC into the set H of sets of dependent constraints. More specifically, function GETDEPENDENTSETSOFCSTRS identifies the connected sub-hypergraphs (i.e., the maximal connected components) of the hypergraph equivalent to the constraint network [107, 103] representing the attack condition; each set $H_i \in H$ corresponds to a sub-hypergraph. For the attack condition $AC = SEC0$ in our example, we have $H = \{H_1, H_2, H_3, H_4\}$ with $H_1 = \{cstr_1\}$, $H_2 = \{cstr_2\}$, $H_3 = \{cstr_3\}$, $H_4 = \{cstr_4, cstr_5, cstr_6\}$ where:

$$
\begin{aligned}
cstr_1 \equiv & SUBJ\texttt{.substring(0,2).equals("cd")} \\
cstr_2 \equiv & \texttt{Integer.parseInt(}MAX\texttt{)} = \texttt{20} \\
cstr_3 \equiv & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \\
cstr_4 \equiv & SID\texttt{.length()} \leq \texttt{20} \\
cstr_5 \equiv & SID\texttt{.contains("id")} \\
cstr_6 \equiv & \texttt{ESAPI.encoder().encodeForXPath(}SID\texttt{)} \\
& \texttt{.matches(".* or 1=1 .*")}
\end{aligned}
$$

Next, the algorithm iterates through the sets of constraints H_i in H, performing the following steps (Lines 5–16).

First, it calls function EXTERNALSOLVE (Line 6), which invokes the external solver and returns a tuple $\langle externalSolved, Sol \rangle$. If the external solver supports all the operations used in the constraints contained in H_i, it will return a true value for the flag $externalSolved$ and the set Sol will contain a solution for each variable involved in the constraints in H_i; the algorithm can then proceed to the next iteration of the loop, to process the set H_{i+i}. Otherwise, in case the external solver does not support an operation used in a constraint in H_i, it will fail and EXTERNALSOLVE will set the flag $externalSolved$ to false and Sol to the empty set.

When the flag $externalSolved$ is false, the algorithm enters the second stage of our approach. It calls function AUTOMATASOLVE (Line 8), which internally invokes the automata-based string constraint solver *Sushi* [40] to solve the constraints in H_i that use operations it supports (i.e., `concat`, `contains`, `equals`, `trim`, `substring`, `replace`, `replaceAll`, and `matches`). If a constraint is satisfiable, *Sushi* yields a solution automaton for each string variable involved in the constraint. Function AUTOMATASOLVE returns a tuple $\langle H'_i, Sol \rangle$. The set $H'_i \subseteq H_i$ contains the constraints in H_i that could not be solved by *Sushi* because they use unsupported operations. The set Sol contains the solution automata for the variables involved in the constraints in H_i: if a variable was involved only in constraints with unsupported operations, its corresponding solution automaton is the default one, accepting any string (i.e., the automaton accepting the regular language `.*`); otherwise, the solution automaton is the one determined by *Sushi*. Notice that both EXTERNALSOLVE and AUTOMATASOLVE internally terminate the entire constraint solving procedure and return UNSAT, without proceeding to the following steps, when they detect unsatisfiable constraints.

Subsequently, if the set H_i' is not empty (meaning that there are unsolved constraints in H_i using unsupported operations), the algorithm invokes the SEARCHSOLVE function, which implements a meta-heuristic search algorithm (detailed in the next subsection) to solve the constraints in H_i' and returns an updated set of solutions Sol (Line 10). If the set Sol is not empty, it means that the set of constraints H_i has been solved and the algorithm can proceed to process the set H_{i+i}; otherwise, it means that the SEARCHSOLVE function timed out and thus the algorithm returns TIMEOUT, terminating the entire constraint solving procedure. A time-out can indicate either that a solution exists for the constraint but the solver could not find it, or that the constraint is actually unsatisfiable; the security analyst then has to decide (possibly based on empirical studies) how to treat it.

The algorithm returns SAT and the set of solutions Sol (Line 17) only when the loop over H has been completely executed, meaning that all the constraints in the sets in H are satisfiable, which is equivalent to say that the attack condition AC in input is satisfiable.

For our example attack condition $SEC0$, the call to H_1 will return $\langle true, \{SUBJ = \texttt{"cd"}\}\rangle$, meaning that the external solver was able to solve the constraint, determining the solution "`cd`" for variable $SUBJ$. However, the call to EXTERNALSOLVE with input H_2 and H_3 will return $\langle false, \emptyset\rangle$ because the external solver cannot handle some of the operations used in the constraints in H_2 and H_3 (e.g., the operations `parseInt`, `equalsIgnoreCase`). This means that the constraints in H_2 and H_3 will be processed in the second stage. In particular, the calls to AUTOMATASOLVE will behave as follows. AUTOMATASOLVE$(H_2) = \langle\{cstr_2\}, \{M_{MAX}\}\rangle$, with $M_{MAX} =$ `.*`, meaning that $cstr_2$ could not be solved by *Sushi* (because it contains an unsupported operation). The same is true for $cstr_3$ with AUTOMATASOLVE$(H_3) = \langle\{cstr_3\}, \{M_{OP}\}\rangle$, with $M_{OP} =$ `.*` and AUTOMATASOLVE$(H_4) = \langle\{cstr_4, cstr_5, cstr_6\}, \{M_{SID}\}\rangle$, where $M_{SID} =$ `.*id.*`, meaning that *Sushi* could only solve $cstr_5$ and determine a solution automaton for variable SID.

4.3.2 Solving Constraints using Meta-heuristic Search

We use the ACO meta-heuristic search for solving constraints that involve unsupported operations. We chose ACO over other well-known meta-heuristic search techniques (such as hill climbing, simulated annealing, and genetic algorithms [48]) because:

- It is typically used for finding good solutions (i.e., paths that return good fitness values) in graphs [36]. Hence, it can be easily adapted to our problem where the search space is defined in a graph form, i.e., an automaton.
- Differently from other search algorithms, in ACO, defining a new candidate solution is straightforward, since it only requires having an ant exploring a new path in the solution automaton.
- It has inherent parallelism in which multiple candidate solutions can be searched in parallel for efficiency.
- It is stochastic in nature, which prevents the search from getting stuck in local optima.

Several algorithms that implement the ACO meta-heuristic have been proposed in the literature. In this work we will use $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [122] with 2-opt local search [31], in

which the pheromone values are bounded by maximum and minimum values, which are dynamically computed after every search iteration. We use this algorithm because the bounding of the pheromone values prevents their relative difference from becoming too extreme during the run of the algorithm and, therefore, mitigates the search stagnation problem in which ants traverse the same trails and construct the same solutions over and over again.

Below, we first present the fitness functions (Section 4.3.2.1) used within the algorithm and then the algorithm (Section 4.3.2.2) itself.

4.3.2.1 Fitness Functions

Any search-based procedure requires defining one or more fitness functions to assess the quality of the potential solutions, i.e., their distance from the best solution. A low(er) value for the fitness of a solution implies a high(er) quality for the solution itself. Since in the context of this work we deal with both integer and string constraints, we use fitness functions specific to these domains.

For integer constraints we use the Korel function [66], which is a standard fitness function for this domain. We consider constraints of the form $C \equiv E_1 \bowtie E_2$, where $\bowtie \in \{=, <, \leq, >, \geq\}$ and E_1, E_2 are integer expressions that can be integer variables, integer constants, or any other expression whose evaluation results in an integer value (e.g., the `length` operation for strings); notice that we treat boolean expressions also as integer expressions. Let $\mathbf{s} = [s_1, \ldots, s_n]$ be the vector of candidate solutions for the integer variables $x_1, \ldots, x_n$ in C, and $a(\mathbf{s}), b(\mathbf{s})$ be the integer values resulting from the evaluations of E_1 and E_2 respectively, after replacing the variables in them with the corresponding solutions in $\mathbf{s}$. The fitness of $\mathbf{s}$ is defined as:

$$f(\mathbf{s}) = \begin{cases} 0, & a(\mathbf{s}) \bowtie b(\mathbf{s}) \text{ is true;} \\ |a(\mathbf{s}) - b(\mathbf{s})| + k, & a(\mathbf{s}) \bowtie b(\mathbf{s}) \text{ is false;} \end{cases} \tag{4.3.1}$$

where $k = 0$ when $\bowtie \in \{=, \leq, \geq\}$ and $k = 1$ otherwise.

For string constraints we use two different functions, depending on the operations in which string variables are involved: the Levenshtein (edit) distance function [89] and the equality cost function for regular expression matching [4]; both functions have been shown to be useful for search-based generation of string values [4]. The Levenshtein distance between two strings a and b is defined as the minimum number of insert, delete, and substitute operations (of characters) needed to convert a into b. The regular expression matching function between a string a and a regular expression b is defined as the minimum Levenshtein distance among a and the strings belonging to the regular language defined by b. We consider string constraints of the form $C \equiv E_1 \bowtie E_2$, where $\bowtie$ is a string operation returning a boolean result, and E_1, E_2 are string expressions that can be string variables, string literals, or any other expression whose evaluation results in a string value (e.g., the `concat` operation for two strings). Let $\mathbf{s} = [s_1, \ldots, s_n]$ be the vector of candidate solutions for the string variables $x_1, \ldots, x_n$ in C, and $a(\mathbf{s}), b(\mathbf{s})$ be the string values resulting from the evaluations of E_1 and E_2 respectively, after replacing the variables in them with the corresponding solutions in $\mathbf{s}$. The fitness of $\mathbf{s}$ is defined as:

$$f(\mathbf{s}) = \begin{cases} 0, & a(\mathbf{s}) \bowtie b(\mathbf{s}) \text{ is true;} \\ \psi\left(a(\mathbf{s}), b(\mathbf{s})\right), & a(\mathbf{s}) \bowtie b(\mathbf{s}) \text{ is false;} \end{cases} \tag{4.3.2}$$

where ψ is:

- the equality cost function for regular expression matching, when $\bowtie$ is a regular expression-based string matching operation (e.g., the `matches` operation for strings in Java);
- the Levenshtein distance, in all other cases for $\bowtie$.

We assume to have a list of operations classified as regular expression-based string matching operations; if there is an unknown regular expression-based matching operation, it will be treated as a generic case, using the Levenshtein distance function. For both types of constraints, the fitness of a candidate solution is set to an arbitrarily selected large value (such as 1000) when the solution leads to an exception during the evaluation of the expressions in which it is used.

We now show the application of these fitness functions in the context of solving the constraints used in the attack condition *SEC0* from our example (see Section 4.2). *SEC0* contains two integer constraints:

$$cstr_2 \equiv \texttt{Integer.parseInt(}MAX\texttt{)} = \texttt{20}$$
$$cstr_4 \equiv SID\texttt{.length()} \leq \texttt{20}$$

one string constraint with a regular expression-based string matching operation:

$$cstr_6 \equiv \texttt{ESAPI.encoder().encodeForXPath(}SID\texttt{)}$$
$$\texttt{.matches(".* or 1=1 .*")}$$

and two constraint with generic string operation:

$$cstr_3 \equiv OP\texttt{.trim().equalsIgnoreCase("GradeQuery")}$$
$$cstr_5 \equiv SID\texttt{.contains("id")}$$

Note that $cstr_1$ is omitted here because it has been already solved as SAT by the external constraint solver (see Section 4.3.1).

Let us consider the case in which the search algorithm has returned the following candidate solutions for the variables involved in *SEC0*: $MAX \coloneqq 20$, $OP \coloneqq$ `Grade` and $SID \coloneqq$ `123id or1`, the fitness function for these solutions is computed as follows.

For constraint $cstr_2$ and $cstr_4$, we apply Eq. 4.3.1; since the evaluation of the constraint $cstr_2$ after replacing the variable with the candidate solutions are true. For example in the case of $cstr_2$, we get $f(MAX \coloneqq 20) = |20 - 20| = 0$.

For constraint $cstr_6$, we apply Eq. 4.3.2; since the evaluation of the constraint after replacing the variable with the candidate solution is false (because the string `123id or1` does not match the regular expression `".* or 1=1 .*"`), we get:

$$f(SID \coloneqq \texttt{123id or1}) =$$
$$\psi(\texttt{123id or1}, \texttt{".* or 1=1 .*"}) = 4$$

In this case, ψ is the equality cost function for regular expression matching; a fitness value equal to 4 means that at least four character operations are needed to convert the candidate solution to a string belonging to the regular language defined by the given regular expression. Following a similar process, the fitness of the candidate solution for OP in the constraint $cstr_3$ is computed as $f(OP \coloneqq \texttt{Grade}) = \psi(\texttt{Grade}, \texttt{GradeQuery}) = 5$, where ψ is the Levenshtein distance function. For constraint $cstr_5$, we get $f(SID \coloneqq \texttt{123id or1}) = 0$ because the solution for SID satisfies the constraint.

```
1: function SEARCHSOLVE(
   Hypergraph H,
   Set of Solution Sol)
2:     Set of Solution-automaton K ←
       GETSOLAUTOMATA(Sol)
3:     Tuning-parameters ⟨α, β, ρ, ξ_max, ξ_min⟩ ←
       SETTUNINGPARAMS()
4:     Population-size A ← SETNUMBERANTS()
5:     Set of Desirability-value Δ ← SETDESIRABILITYVAL(K)
6:     Set of Pheromone Ξ ← SETPHEROMONES(K)
7:     Set of Solution-component T_Best ← ∅
8:     Fitness F_Best ← 1; Fitness F_pBest ← 1
9:     Array of Fitness tempF ← ∅
10:    Array of Set of Solution-component tempT ← ∅
11:    repeat
12:        loop A times
13:            Set of Solution-component T ←
               CONSTRUCTSOLUTIONS(K, Δ, Ξ)
14:            Fitness F ← COMPUTEFITNESS(T, H)
15:            tempF ← APPEND(tempF, F)
16:            tempT ← APPEND(tempT, T)
17:        end loop
18:        ⟨F_Best, T_Best⟩ ← BESTSOLUTION(tempF, tempT)
19:        if F_Best < F_pBest then
20:            ⟨F_Best, T_Best⟩ ← 2OPTLOCALSEARCH(K, T_Best)
21:        end if
22:        UPDATEPHEROMONES(K, Ξ, F_Best, T_Best)
23:        F_pBest ← F_Best
24:    until F_Best = 0 or timeout
25:    if timeout then
26:        return ∅
27:    end if
28:    Sol ← UPDATESOL(T_Best)
29:    return Sol
30: end function

31: function CONSTRUCTSOLUTIONS(
    Set of Solution-automaton K,
    Set of Desirability-value Δ,
    Set of Pheromone Ξ)
32:    Set of Solution-component S ← ∅
33:    repeat
34:        Automaton k ← RANDOMSELECT(K)
35:        FSMState v ← GETSTARTSTATE(k)
36:        repeat
37:            Set of FSMTransition E ←
               GETOUTTRANSITIONS(v)
38:            FSMTransition e ←
               SELECTTRANSITION(E, Δ, Ξ)
39:            S ← S ∪ {e}
40:            v ← GETNEXTSTATE(e)
41:        until ISACCEPTSTATE(v)
42:        MARKASVISITED(k, K)
43:    until all the automata in K have been traversed
44:    return S
45: end function

46: function COMPUTEFITNESS(
    Hypergraph H,
    Set of Solution-component T)
47:    Set of Constraint Θ ← GETCONSTRAINTS(H)
48:    i ← 0
49:    for all Constraint θ in Θ do
50:        i ← i + 1
51:        Fitness f_i ← EVALUATE(θ, T)
52:        Fitness f̂_i ← NORMALIZE(f_i)
53:    end for
54:    return Fitness F ← AVERAGE(f̂_1, f̂_2, ..., f̂_i)
55: end function
```

Algorithm 4.5: Ant colony search for string constraint solving.

4.3.2.2 Search Algorithm

The ACO meta-heuristic for solving constraints containing unsupported operations is implemented in function SEARCHSOLVE, whose pseudocode is shown in Algorithm 4.5. The function takes as input a hypergraph H and a set of solutions Sol.

First, the function retrieves from Sol the set of solution automata K for the string input variables (Line 2); notice that auxiliary string variables are excluded because the search procedure needs to find solutions only for the input variables. The next steps of function SEARCHSOLVE (Lines 3–6) initialize the tuning and search parameters as follows (the initialization value is indicated next to each parameter):

- *Tuning parameters*: $\alpha = 1$ and $\beta = 1$ determine the relative importance of the pheromone trail and the heuristic-based desirability information; $\rho = 0.01$ is the evaporation rate used to prevent the pheromone values from piling up; $\xi_{max} = 5$ and $\xi_{min} = 0$ determine the bounds of pheromone values.

- *Search parameters*: the number of ants $A = 20$; the set Δ of desirability values $\delta_e = 1$ for

each transition e of each automaton in K; the set Ξ of pheromone values $\xi_e = \xi_{max}$ for each transition e of each automaton in K.

In ACO, these parameters have to be defined specifically for the target problem; we chose them based on the guidelines provided in [122] and on our own preliminary experiments. Notice that for each transition e, the parameter ξ_e is initialized to the value ξ_{max}; as discussed in [122], this allows for diverse explorations of the solutions during the first iterations of the algorithm, because of the small, relative differences between the pheromone values of the explored transitions and of the ones not-yet explored.

The algorithm then loops through the three main steps (construction of candidate solutions, application of local search, update of pheromone values) until the termination conditions are met (Lines 11–24). We illustrate these steps through the example shown in Figure 4.2. In this example, the variable V is involved in three constraints (shown in Figure 4.2): $cstr_1 \equiv$ `V.matches("ab*|ca")` contains one supported operation; $cstr_2 \equiv$ `V.length()` ≤ 3 contains two supported operations `length` and $\leq$; $cstr_3 \equiv$ `custom(V).equals("bba")` contains one supported operation `equals` and one unsupported operation `custom`.

Construction of candidate solutions. This step (Lines 12–18) consists of three sub-steps:

1. *Building the set of solution components*. This step is represented by the call to function CONSTRUCTSOLUTIONS, which takes as input the set of solution automata K, the set of desirability values Δ, and the set of pheromones values Ξ. It outputs a set of solution components; a solution component is a sequence of transitions in a solution automaton as traversed by the procedure.

   This function goes through (Lines 33–43) the set of automata K, and at each iteration it randomly selects an automaton $k \in K$. Starting from the start state of k, it traverses the outgoing transitions of the states in k[2].

   Upon reaching a state where there are multiple outgoing transitions, it selects (Line 38) one of them (say transition e) based on the probability $P_e = \frac{\xi_e^\alpha \delta_e^\beta}{\sum_{t \in E} \xi_t^\alpha \delta_t^\beta}$, computed using the pheromone value ξ_e and the desirability δ_e of the transition.

   The selected transition is added to the local set of solution components S (Line 39) and its reaching state is retrieved (Line 40). The traversal/selection of the transitions of an automaton is repeated until the final state is found[3], which means that a solution for the variable associated with the current automaton k has been found. In this case the outer loop moves to explore the next automaton in K, and continues until all automata in K have been traversed. At the end, the function returns a set of solution components, with one candidate solution for each string variable.

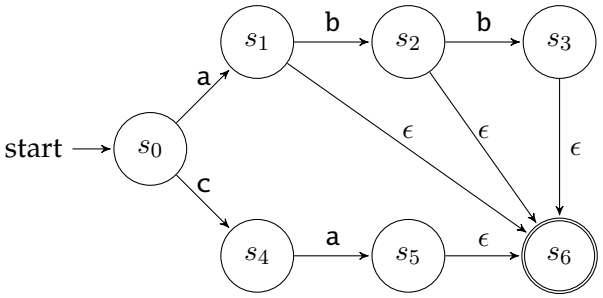
   Let us execute this step through the example shown in Figure 4.2, by assuming that the automaton k extracted at Line 34 is the one shown in Figure 4.2b. This is the solution automaton of the input variable V, as determined after solving the first two constraints with

[2] In our automaton representation, a transition reflects a Unicode character; each transition is updated with a new pheromone value during the search iterations to reflect the fitness of the solution that contains the corresponding character.

[3] Internally we represent solution automata as generalized non-deterministic finite automata, which have only one final state.

V.matches("ab*|ca")
$\wedge V$.length() ≤ 3
$\wedge$ custom(V).equals("bba")

(a) Three constraints in which variable V is involved.



(b) The solution automaton of V for the first two constraints, representing the regular language $(ab^*|ca) \cap .\{0,3\}$

Figure 4.2: Example to illustrate the search algorithm.

the procedure described in Section 5.3.3.1; each transition is associated with a character or the empty string ϵ. We assume that the transition (s_0, s_1) has the pheromone value $\xi_{(s_0,s_1)} = 2$, while the other transitions have the pheromone value $\xi = 1$; also recall that the desirability value for all the transitions is set to a fixed value of $\delta = 1$.

To construct a solution, the procedure starts from the initial state s_0 and then selects one of its outgoing transitions based on the probability P_e. In this case, the probability of selecting transition (s_0, s_4) is $P_{(s_0,s_4)} = (1 * 1)/(1 + 2) = 0.33$; the probability of selecting transition (s_0, s_1) is $P_{(s_0,s_1)} = (2 * 1)/(1 + 2) = 0.67$. Let us assume that the transition (s_0, s_1) is selected and traversed: the procedure reaches state s_1. It then traverses one of the two outgoing transitions of s_1; the probability of selecting transition (s_1, s_6) is 0.5 and the probability of selecting transition (s_1, s_2) is 0.5. Assuming that the procedure selects (s_1, s_2), it reaches state s_2. Afterwards, assuming that the procedure selects transition (s_2, s_6), it reaches the final state s_6. This generates the sequence of transitions $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$, which represents the candidate solution $V =$"ab".

2. *Determining the fitness of solution components.* This step computes the fitness for the solution components identified in the previous step. Function COMPUTEFITNESS (Line 14) takes as input the hypersubgraph H and the set of solution components T, which contains a candidate solution for each input variable in H. It first obtains the set of constraints Θ represented by H. Then, for each constraint θ in Θ it invokes[4] function EVALUATE (Line 51), which evaluates the expressions in θ with the solutions given in T. More specifically, each unsupported operation in θ is invoked with parameters that are retrieved from T; the return value is then used to compute the fitness f, using one of the aforementioned fitness

[4] To invoke unsupported operations, we assume that the corresponding bytecode is accessible through the classpath; we use the Java reflection methods to load and execute the code of the unsupported operation. Notice that this execution is subject to the timeout defined in function CSTRSOLVE.

functions (Eq. 4.3.1 or Eq. 4.3.2) depending on the type of constraint, as explained in Section 4.3.2.1. To ensure that the search process is not biased towards solving the constraints with larger-scale fitness values, each fitness value f is normalized (Line 52) using the normalization function proposed in [12], resulting in a normalized fitness value $\hat{f} = f/(f+1)$. We use this normalization function since it has been proven to be useful in the similar domain of search-based test input generation of string data types [82]. After computing the fitness for all the constraints in H, the overall fitness F of T is computed by taking the average of individual, normalized fitness values $\hat{f}$ (Line 54).

The execution of this step through the example in Figure 4.2 works as follows. Recall that the solution identified in the previous step is $V =$ `"ab"`. For each constraint, the corresponding expression is evaluated and the fitness of the solution is computed accordingly. For $cstr_1$, we apply Eq. 4.3.2 with the equality cost function for regular expression matching; since the evaluation of the constraint after replacing the variable with the candidate solution is true, we get $f_{cstr_1} = 0$. For $cstr_2$, we apply the Korel function (Eq. 4.3.1); since the evaluation of the constraint after replacing the variable with the candidate solution is true, we get $f_{cstr_2} = 0$. For constraint $cstr_3$, let us assume that the resulting value after executing the operation `custom` with the input $V =$ `"ab"` is `"ba"`; the fitness, computed by applying the Levenshtein (edit) distance function, is $f_{cstr_3} = 1$[5]. By applying the normalization we get:

$$\hat{f}_{cstr_1} = f_{cstr_1}/(f_{cstr_1} + 1) = 0$$
$$\hat{f}_{cstr_2} = f_{cstr_2}/(f_{cstr_2} + 1) = 0$$
$$\hat{f}_{cstr_3} = f_{cstr_3}/(f_{cstr_3} + 1) = 1/(1 + 1) = 0.5$$

Finally, the overall fitness F of the solution $V =$`"ab"` is computed as:

$$F = avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}, \hat{f}_{cstr_3}) = 0.16$$

3. *Selecting the best solution components.* The two steps above are repeated A times, with the values computed at each iteration stored as elements of the auxiliary variables $tempT$, an array containing sets of solution components, and $tempF$, an array containing the fitness values for the corresponding elements in $tempT$. Function BESTSOLUTION determines among them the solution components that have the minimum (i.e., best) fitness, and assign them to variable T_{Best}, representing the best solution of the current iteration of the outer loop.

Application of local search. This step (Lines 19–21) is used to refine the set of candidate solutions built in the step above, to locally optimize them. More precisely, if the best solution of the current iteration (T_{Best}) is better than (i.e., its fitness is lower than the fitness of) the best solution of the previous iteration, we perform a local search procedure to see whether further improvements can be made with other solutions that are in the neighborhood of T_{Best}. The local search is performed using the 2-opt local search algorithm [76], which

[5]This means that the insertion, modification, or deletion of one character is required in order to satisfy this constraint (see Section 4.3.2.1).

finds in each automaton in K other paths (or sequence of transitions) that reach the final state. This algorithm replaces at most two transitions of the current path with one or more transitions; if it finds a set of solution components with a better fitness value, this set becomes the new T_{Best}.

To illustrate this step through the example in Figure 4.2, let us assume that the current best solution T_{Best} is the sequence of transitions $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$, which represents the solution $V =$"ab". The application of 2-opt local search algorithm might result in replacing the second transition (s_1, s_2) with a different transition (s_1, s_6). This produces a new sequence of transitions $\{(s_0, s_1), (s_1, s_6)\}$, which represents a new candidate solution $V =$"a". The fitness of this new solution is lower than the one of the current best solution and, hence, the current T_{Best} is not changed. This example shows that a local search procedure may not always find a better solution; nevertheless it is useful when there is a better solution in the neighborhood of the current search space.

Update of pheromone values. This step (Line 22) updates the pheromone values $\xi_e \in \Xi$, for each transition e of each automaton in K. It first computes $\xi_{max} = \frac{1}{1-\rho}\frac{1}{F_{Best}}$ and $\xi_{min} = \frac{\xi_{max}}{2n}$, where n denotes the cumulative total number of states of all the automata in K; then, it sets $\xi_e = (1-\rho)\xi_e + \triangle\xi_e$, where $\triangle\xi_e = \frac{1}{F_{Best}}$ if the transition e is part of the solution components in T_{Best}, 0 otherwise. If $\xi_e > \xi_{max}$, then it sets $\xi_e = \xi_{max}$; dually, if $\xi_e < \xi_{min}$, then it sets $\xi_e = \xi_{min}$.

In the case of the example in Figure 4.2, recall that the current best solution T_{Best} is the sequence of transitions $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$, with $F_{Best} = 0.16$. The pheromone values of the transitions of the solution automaton are updated as follows:

$$
\begin{aligned}
\xi_{max} &= \frac{1}{1-\rho}\frac{1}{F_{Best}} = \frac{1}{1-0.01}\frac{1}{0.16} = 6.31 \\
\xi_{min} &= \frac{\xi_{max}}{2n} = 6.31/(2*7) = 0.45 \\
\xi_{(s_0,s_1)} &= (1-\rho)\xi_{(s_0,s_1)} + \triangle\xi_{(s_0,s_1)} \\
&= (1-0.01)*2 + 1/0.16 = 8.23 \rightarrow 6.31 \\
\xi_{(s_1,s_2)} &= (1-0.01)*1 + 1/0.16 = 7.24 \rightarrow 6.31 \\
\xi_{(s_2,s_6)} &= (1-0.01)*1 + 1/0.16 = 7.24 \rightarrow 6.31 \\
\xi_{(s_0,s_4)} &= (1-0.01)*1 + 0 = 0.99 \\
\xi_{(s_4,s_5)} &= (1-0.01)*1 + 0 = 0.99 \\
\xi_{(s_5,s_6)} &= (1-0.01)*1 + 0 = 0.99 \\
\xi_{(s_1,s_6)} &= (1-0.01)*1 + 0 = 0.99 \\
\xi_{(s_2,s_3)} &= (1-0.01)*1 + 0 = 0.99 \\
\xi_{(s_3,s_6)} &= (1-0.01)*1 + 0 = 0.99
\end{aligned}
$$

Regarding the transitions (s_0, s_1), (s_1, s_2), and (s_2, s_6), their pheromones values are set to ξ_{max} since their originally computed values are larger than ξ_{max}. Note that, for the

transitions $(s_0, s_4), (s_4, s_5), (s_5, s_6), (s_1, s_6), (s_2, s_3), (s_3, s_6)$, $\triangle\xi = 0$ since these transitions are not part of T_{Best}.

The termination conditions of the loop at Line 24 in Algorithm 4.5 correspond either to a time-out or to the finding of a solution that satisfies all the constraints in H, for which the fitness F_{Best} is zero. If there is a timeout, the function returns an empty set of solutions. Otherwise, it updates Sol with T_{Best} (Line 28); i.e., the solution automata in Sol are replaced with the solutions represented by T_{Best} for the corresponding variables, and it returns Sol.

4.4 Implementation

We have implemented our search-driven string constraint solving approach for vulnerability detection in the *ACO-Solver* tool [128]. The tool is implemented in Java, uses *Sushi* as automata-based constraint solver in the second stage, and has a plugin architecture to support different solvers in the first stage; we have developed plugins for *CVC4* and *Z3-str2*.

4.5 Evaluation

In this section we report on the evaluation of *ACO-Solver* in the context of vulnerability detection for Java Web applications. We assess the benefits and costs of combining the proposed string constraint solving approach with two state-of-the-art solvers, by answering the following research questions:

RQ1 How does the proposed approach improve the effectiveness of state-of-the-art solvers for solving constraints related to vulnerability detection? (Section 4.5.2)

RQ2 Is the cost (in terms of execution time overhead) of using our technique affordable in practice? (Section 4.5.2)

RQ3 Does the automata-based solver in the first step of the second stage of our approach contribute to the effectiveness of the search-based procedure? (Section 4.5.3)

4.5.1 Benchmark and Evaluation Settings

To evaluate our approach in terms of vulnerability detection capability, we use the same benchmark as presented in Section 3.4 composed of nine realistic, open source Java Web applications/services, with known XSS, XML, XPath, LDAP, and SQL injection vulnerabilities.

This benchmark contains in total 104 paths to sinks: 64 vulnerable paths and 40 non-vulnerable ones. We generated the corresponding 104 attack conditions using our security slicing approach presented in Chapter 3. For each attack condition, we established the ground truth (i.e., whether it is vulnerable or not) via manual inspection and consultation of the vulnerability report of the corresponding application in the National Vulnerability Database (NVD) [91].

We conducted our evaluation on a machine equipped with an Intel Core i7 $2.4\,\mathrm{GHz}$ processor, $8\,\mathrm{GB}$ memory, running Apple Mac OS X 10.11 and *Sushi* v2.0. We set the time-out for solving each attack condition to $30\,\mathrm{s}$.

Table 4.1: Comparison of vulnerability detection effectiveness and execution time between standalone solvers (*Z3-str2* and *CVC4*) and the same solvers *combined* with *ACO-Solver* (*Z3-str2 + ACO-Solver* and *CVC4 + ACO-Solver*).

App	Paths		*Z3-str2*								*Z3-str2 + ACO-Solver*									*CVC4*							*CVC4 + ACO-Solver*								
	vp	*nvp*	t(s)	X	✓	*tp*	*tn*	*fp*	*fn*	*pd*	t(s)	⊘	✓	Δ	*tp*	*tn*	*fp*	*fn*	∇	*pd*	t(s)	X	✓	*tp*	*tn*	*fp*	*fn*	*pd*	t(s)	⊘	✓	Δ	*tp*	*tn*	*fp*	*fn*	∇	*pd*
WebGoat	11	4	0.1	11	4	0	4	0	11	0	22.2	0	15	11	11	4	0	0	11	100.0	1.4	1	14	10	4	0	1	90.9	10.9	0	15	1	11	4	0	0	1	100
Roller	3	10	0.0	13	0	0	10	0	3	0	334.0	10	3	3	3	10	0	0	3	100.0	0.5	10	3	3	10	0	0	100.0	307.2	10	3	0	3	10	0	0	0	100
Pebble	6	7	0.0	12	1	0	7	0	6	0	199.3	5	8	7	6	7	0	0	6	100.0	0.0	12	1	0	7	0	6	0.0	205.1	5	8	7	6	7	0	0	6	100
Regain	3	3	86.7	0	6	3	3	0	0	100	84.1	0	6	0	3	3	0	0	0	100.0	0.6	0	6	3	3	0	0	100.0	1.5	0	6	0	3	3	0	0	0	100
PSH	1	3	13.3	4	0	0	3	0	1	0	61.6	2	2	2	1	3	0	0	1	100.0	0.0	4	0	0	3	0	1	0.0	61.3	2	2	2	1	3	0	0	1	100
RAP	1	0	0.0	1	0	0	0	0	1	0	0.4	0	1	1	1	0	0	0	1	100.0	0.0	1	0	0	0	0	1	0.0	0.9	0	1	1	1	0	0	0	1	100
TPC-APP	6	6	0.0	10	2	0	6	0	6	0	217.8	7	5	3	2	6	0	4	2	33.3	0.6	3	9	6	6	0	0	100.0	93.2	3	9	0	6	6	0	0	0	100
TPC-C	30	4	0.1	31	3	0	4	0	30	0	596.4	15	19	16	16	4	0	14	16	53.3	1.5	1	33	30	4	0	0	100.0	47.1	1	33	0	30	4	0	0	0	100
TPC-W	3	3	0.0	3	3	0	3	0	3	0	2.4	0	6	3	3	3	0	0	3	100.0	0.3	0	6	3	3	0	0	100.0	1.2	0	6	0	3	3	0	0	0	100
Total	64	40	100.3	85	19	3	40	0	61	5	1,518.3	39	65	46	46	40	0	18	43	71.9	5.0	32	72	55	40	0	9	85.9	728.6	21	83	11	64	40	0	0	9	100

4.5.2 Effectiveness and Cost of Vulnerability Detection

We assess the benefits and costs of combining the proposed approach with two state-of-the-art string constraint solvers: *CVC4* (version 1.4) and *Z3-str2* (from the repository head, commit 2e52601). For each of these solvers, we run our benchmark first through the standalone solver and then through the solver *combined* with *ACO-Solver*.

The evaluation results are shown in Table 4.1. Columns *vp* and *nvp* indicate, respectively, the number of vulnerable and non-vulnerable paths per application. Column t indicates the cumulative time taken to solve all the attack conditions of each application. Column X indicates the number of failing cases, i.e., the number of attack conditions that the solver failed to solve, due to an error or crashing; we omit this column for *Z3-str2 + ACO-Solver* and *CVC4 + ACO-Solver* since they did not fail. Column ⊘ indicates the number of cases in which the solver timed out; we omit this column for *Z3-str2* and *CVC4* since they did not time out. Column ✓ indicates the number of non-failing cases. Column Δ indicates the number of cases, out of the failing cases of *Z3-str2* or *CVC4*, that *ACO-Solver* helped solve. Columns *tp*, *tn*, *fp*, *fn*, and ∇ denote, respectively, true positives (number of vulnerable cases correctly identified), true negatives (number of non-vulnerable cases correctly identified), false positives (number of non-vulnerable cases reported as vulnerable), false negatives (number of vulnerable cases not detected), number of additional vulnerable cases uncovered by *ACO-Solver*. Column *pd* reports the *recall*, i.e., the percentage of vulnerable cases detected among the total vulnerable cases, and is computed as $pd = tp/(tp + fn) * 100$. Notice that, in the context of vulnerability detection, when a solver fails or times out to solve an attack condition, it neither detects a vulnerability nor produces a false alarm. Hence, a failing (or time-out) case may result either in a false negative or in a true negative, depending on whether the attack condition is actually vulnerable.

We answer *RQ1* by examining the number of failing and time-out cases and the recall in Table 4.1, first when using a solver standalone, and then when combined with *ACO-Solver*.

When used standalone, *Z3-str2* and *CVC4* could not solve, respectively, 85 and 32 cases. We manually inspected these failing cases and observed they are due to unsupported operations contained in the attack conditions. For example, both *Z3-str2* and *CVC4* do not handle some string operations (e.g., `toLowerCase`, `toUpperCase`, `equalsIgnoreCase`) and the sanitiza-

tion operations of the standard Apache security library [8]. Also, *Z3-str2* was not able to handle `Integer.parseInt`, and `String.valueOf` conversions and many of the regular expressions that reflect security threats in our attack conditions. *Z3-str2* missed 61 vulnerable cases (out of 85 failing cases), resulting in a low recall of 4.7%. *CVC4* missed 9 vulnerable cases (out of 32 failing cases), resulting in a recall of 85.9%.

Z3-str2 + ACO-Solver helped solve 46 out of the 85 failing cases of *Z3-str2*, revealing 43 additional vulnerabilities. It timed out on 39 cases; however, 21 out of these 39 time-out cases are non-vulnerable cases (i.e., the corresponding attack condition is UNSAT) and thus the search is obviously expected to time out. *CVC4 + ACO-Solver* solved 11 failing cases of *CVC4*, revealing 9 additional vulnerabilities. It timed out on 21 cases; however, all of them are actually non-vulnerable ones. *Z3-str2 + ACO-Solver* improved the recall of *Z3-str2* from 4.7% to 71.9%. *CVC4 + ACO-Solver* improved the one of *CVC4* from 85.9% to 100.0%, detecting all vulnerabilities.

We remark that, while most of these vulnerabilities had already been reported to NVD [91], we also discovered two new XSS vulnerabilities (one in *Regain* and one in *Pebble*) while performing this evaluation; we reported them to NVD and also to the corresponding developers. The vulnerability in *Regain* was detected by both *Z3-str2* and *CVC4*, used standalone; the one in *Pebble* was detected when both solvers were combined with *ACO-Solver*. Though not shown in Table 4.1, we also remark that both solvers achieved 100% precision (i.e., they reported no false positive).

The answer to *RQ1* is that the proposed approach, when combined with a state-of-the-art solver, significantly improves the recall (from 4.7% to 71.9% for *Z3-str2*, from 85.9% to 100.0% for *CVC4*), and solves several cases on which the solvers failed when used stand-alone (46 more solved cases for *Z3-str2*, and 11 more for *CVC4*). Hence, combining a state-of-the-art solver with our approach proved to be very effective to vulnerability detection. Since time-outs with *CVC4 + ACO-Solver* are all unsatisfiable/non-vulnerable cases, if such results were to be confirmed by additional benchmarks, then one could conclude that the most cost-effective and realistic decision strategy for the security analyst would be to treat time-outs as non-vulnerable cases.

To answer *RQ2*, we compare the execution time (t) for running the standalone solvers to the one for running the solvers combined with *ACO-Solver*. The total execution of *Z3-str2* took less than two minutes and solved 19 cases; *Z3-str2 + ACO-Solver* took about 25 minutes and solved 65 cases; the execution of *CVC4* took about five seconds and solved 72 cases; *CVC4 + ACO-Solver* took about 12 minutes and solved 83 cases. Despite the increase in terms of absolute values, the total execution time of our approach is still affordable, considering that 1) it can handle many cases that would otherwise fail, and thus can detect more vulnerabilities; 2) vulnerability detection is typically an offline activity, with no real-time requirements. Hence, we answer *RQ2* positively.

4.5.3 The Role of the Automata-based Solver

To address *RQ3* and thus investigate the role played by the automata-based solver in reducing the search space explored by the meta-heuristic search, we used a modified implementation of our approach. We switched off the automata-based solver in the first step of the second stage, meaning that the search-based algorithm is executed using a set of solution automata that accept any string and thus has a much larger search space.

We run our benchmark through both solvers, combined with this *modified* version of *ACO-Solver*, which does not call *Sushi* internally. We present only the results of running *CVC4* combined with this modified *ACO-Solver*; the results for *Z3-str2* are similar.

When executed with $30\,\mathrm{s}$ time-out, *CVC4* + modified *ACO-Solver* timed out on 31 cases, with a recall of 87.5% and an overall execution time of $15.5\,\mathrm{min}$. Therefore, the version of *ACO-Solver* without *Sushi* helped solve only 1 more case, whereas the unmodified *ACO-Solver* helped solve 11 more cases. Since in this scenario we expected the search to explore a larger search space, we also ran the solver with an increased time-out of $300\,\mathrm{s}$; however, we obtained the same results as above in terms of solved cases and recall, but the execution time increased to almost three hours.

From the above results, we answer *RQ3* by saying that the automata-based solver plays a fundamental role in achieving a higher effectiveness, since it contributes to reducing the number of failing cases and increasing the recall.

4.5.4 Verifiability and Threats to Validity

Verifiability

The applications composing the benchmark, the related attack conditions, the instructions and scripts to obtain the *ACO-Solver* tool and run the benchmark, and the detailed evaluation results are available on our website [128].

Threats to Validity

Our results are based on solving the constraints corresponding to attack conditions extracted from a specific benchmark; hence, they cannot necessarily be generalized to all types of constraints. We minimized this threat by choosing applications that vary in functionality and by sampling realistic projects, which in many cases represent well-known benchmarks in the context of vulnerability detection. There are other benchmarks (e.g., the one used in [62] and the Kaluza suite [112]) that are widely-used for comparing constraint solvers. However, they are not specific to the security domain (e.g., they are not annotated with vulnerability information), and thus the constraints they contain cannot be used to assess the effectiveness of a solver in terms of vulnerability detection. Furthermore, we remark that our results should be interpreted in the *specific context of vulnerability detection*, and cannot (and do not aim to) be extrapolated to the more general case of string constraint solving.

As shown in Section 4.5.3, the role of an automata-based solver is essential for our approach in order to reduce the input domains and scale the meta-heuristic search process. Instead of *Sushi*, which supports only basic operations, we could use other, more powerful automata-based solvers like Stranger [149] and JST [41], which support more operations. Nevertheless, *Sushi* was available from the authors, fully functional, and yielded a significant reduction in search space that was sufficient to make the approach practical. By using an automata-based solver with support for a larger set of operations, we expect a reduction of the time taken by the meta-heuristic search, since it will have to explore a smaller search domain. Hence, our results should be interpreted as the lowest bound for our search-driven constraint solving approach.

4.6 Summary

This chapter addresses the issue of adding support for (complex) string operations in existing string constraint solvers in the context of vulnerability detection. We have proposed a search-driven constraint solving technique that *complements* the support for complex string operations provided by any existing string constraint solver. This technique uses a hybrid constraint solving procedure based on the ACO meta-heuristic.

We have evaluated the proposed technique in the context of injection and XSS vulnerability detection for Java Web applications. More specifically, we have assessed the benefits and costs of combining the proposed technique with two state-of-the-art string constraint solvers (*Z3-str2* and *CVC4*), on a benchmark with 104 constraints derived from nine realistic Web applications. The experimental results show that the proposed approach, when combined with a state-of-the-art solver, significantly improves the number of detected vulnerabilities (from 4.7% to 71.9% for *Z3-str2*, from 85.9% to 100.0% for *CVC4*), and solves several cases on which the solver fails when used stand-alone (46 more solved cases for *Z3-str2*, and 11 more for *CVC4*); both benefits can be obtained while still keeping the execution time reasonable in practice, in the order of minutes. Furthermore, we have also assessed the role played by the automata-based solver in the search space reduction step that precedes the meta-heuristic search: the results confirm that it contributes to increasing the number of solved cases.

Chapter 5

An Integrated Approach for Injection Vulnerability Analysis

The chapter introduces our integrated approach for injection vulnerability analysis and is organized as follows:

Section 5.1 provides an overview of the approach; Section 5.2 discusses the motivations for this work. Section 5.3 explains the approach in detail; Section 5.4 discusses the implementation Section 5.5 presents the evaluation of the *JOACO* tool; Section 5.6 concludes this chapter.

5.1 Overview

In this chapter, we propose a new analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving.

We leverage our previous work on security slicing (see Chapter 3) to mitigate the path explosion problem, by generating during the symbolic execution only the constraints that characterize the security slices of the program under analysis. This step allows us to identify paths and statements in the program where vulnerabilities can be exploited; this renders the remainder of the approach scalable. The generated constraints are then preprocessed in order to simplify the following step.

The next step uses a hybrid approach that orchestrates a constraint solving procedure for string/mixed and integer constraints with a search-based constraint solving procedure. The idea behind this hybrid solving strategy is to solve a constraint through a two-stage process:

1. First, our solving procedure solves all the constraints with supported operations, by leveraging automata-based solving for solving string and mixed constraints, and linear interval arithmetic for solving integer constraints. In both cases, constraint solving rules are expressed using *recipes* that model the semantics of the operations. In particular, we provide recipes for many string/mixed operations, including 16 input sanitization operations

from widely used security libraries [8, 94], and commonly used integer operations. In this way, the constraints involving supported operations can be efficiently solved, without transforming them into a set of primitive functions.

2. In the second stage, we use the search-driven solving procedure presented in Chapter 4. This procedure leverages the ACO meta-heuristic to solve the remaining constraints which contain unsupported operations. The search space of this procedure is represented by the input domains as determined in the first stage; the search is driven by different fitness functions, depending on the type of the constraints.

The solver in the first stage is used to reduce (possibly in a significant way) the search space, i.e., the domains of the string and integer variables, for the search-driven solving procedure; hence, it makes the search in the second stage more scalable and effective.

This chapter extends the search-driven constraint solving approach presented in Chapter 4 along the following lines:

Hybrid constraint solving technique. The hybrid constraint solving technique described in this chapter and implemented in *JOACO* is a revised and improved version of the one described in Chapter 4 (and implemented in *ACO-Solver*) along the following lines:

- *Constraint pre-processing*. *JOACO* includes a pre-processing step that applies:
  a) *derived constraint generation*, which adds additional constraints to reduce the input domain and solve the constraints more efficiently;
  b) *constraint refinement*, to simplify the constraint network, detect trivially inconsistent constraints, and avoid unnecessary and expensive constraint solving.
- *No dependency on an external solver*. *ACO-Solver*, since it implemented a fallback mechanism, first invoked an external solver (i.e., the solver for which the fallback mechanism was provided) to attempt to solve the input constraint. When this invocation failed (e.g., because the external solver could not solve a constraint with an unsupported operation), *ACO-Solver* had to restart the constraint solving from scratch, since it could not reuse or benefit from any intermediate result determined by the external solver before the failure. In this work, *JOACO* does not depend on any external solver, since it orchestrates the two-stage process sketched above.
- *Built-in support for a larger set of string operations*. *ACO-Solver* relied on the *Sushi* [40] constraint solver to compute solution automata for string constraints, before calling the search-based solving procedure. However, *Sushi* supports very few string operations (`concat`, `contains`, `equals`, `trim`, `substring`, `replace`, `replaceAll`, and `matches`). For *JOACO*, we built our automata-based and interval constraint solver on top of *Sushi* and extended it with support for 38 new operations, including 16 input sanitization operations from the Apache Commons Lang 3 [8] and OWASP [94] standard security libraries. By supporting more operations in our built-in automata-based and interval constraint solver, we are able to minimize the number of invocations to the search-based constraint solving procedure.

Empirical evaluation. This chapter includes a much larger and diverse empirical evaluation. First, we assess the vulnerability detection capability of *JOACO* when analyzing the source

code of Web applications and compare it with state-of-the-art vulnerability detection tools for Java Web applications; this analysis is completely new. Second, we assess the constraint solving capability of our approach by running an experimental study which extends the one presented in Chapter 4 by comparing with different string constraint solvers and by using a larger benchmark collection, which includes four additional benchmarks, used in previous studies, as well as an extended version of our home-grown benchmark.

5.2 Motivation

In this section we highlight the challenges in adopting an approach based on symbolic execution and constraint solving in the context of vulnerability detection. In Section 4.2, we introduced the following three-step procedure for vulnerability detection:

Step 1 Path conditions generation through symbolic execution.

Step 2 Definition of the attack condition.

Step 3 Constraint solving.

However, the execution of this procedure faces two main challenges:

CHL1 *Integrated approach.* Performing the three steps illustrated above requires the integration of program analysis techniques (to identify input sources and sinks, to analyze paths between input sources and sinks, etc.), symbolic execution, definition of security threat models, and constraint solving. This integration is not trivial since it has to be realized keeping in mind that the output of each step has to become the input of the next step. This often requires to pre-process the output of each step, before feeding it to the next one. For example, to avoid the path explosion problem, the symbolic execution step should not explore all the paths of the program, but only those traversing security-sensitive locations (i.e., sources and sinks). Therefore, symbolic execution has to be complemented by techniques such as security slicing. Similarly, the attack conditions could benefit from a simplification step, to speed-up the constraint solving phase.
Furthermore, the majority of existing approaches focus only on one of these steps. For instance, *CVC4* and *Z3-str3* only focus on constraint solving and assume that constraints (including those corresponding to attack specifications) are already available; vulnerability detection approaches such as Andromeda [137], Taj [138], and SFlow [53] only perform static analysis, and do not apply symbolic execution and constraint solving.

CHL2 *Support for complex string operations.* The execution of the first step above assumes that symbolic execution uses a constraint solver that is able to handle string operations like `trim`, `toLowerCase`, `indexOf`, `parseInt`, `equalsIgnoreCase`, and `length`. However, state-of-the-art solvers such as Hampi [63], Kaluza [112], *CVC4* [74], *Z3-str2* [152] and *Z3-str3* [21] do not support at least one of these non-basic, complex operations. When a string operation is not supported by the solver, symbolic execution typically has to analyze the implementation of the operation, to transform it into an equivalent set of basic constraints containing only primitive operations, i.e., operations supported by the solver.

This approach may lead to the path explosion problem and, more generally, scalability issues [24, 117], especially when the implementations of unsupported operations contain many paths.

As explained in Section 4.2, one could modify or enhance an existing solver in order to provide native support for complex operations which is not trivial, or, alternatively, re-express complex operations with their equivalent set of basic constraints that can be solved by the solver which may lead to scalability issues [24, 117].

Another approach [64] to deal with complex string operations relies on dynamic symbolic (concolic) execution [115], and treats complex operations by replacing symbolic values of the inputs involved in those operations with concrete values. However, this approach reduces precision since it can only reason about the paths that are exercised by the concrete values.

To work around this issue, the current solution in practice is to have the constraint solver fail (i.e., it crashes or returns an error) when it encounters an unsupported operation. Our experiments show that this is the case for state-of-the-art solvers like *CVC4* [74] and *Z3-str3* [21]. However, in the context of vulnerability analysis, such a behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

To recap, the two challenges discussed above show that in order to use symbolic execution and constraint solving as effective and scalable techniques for vulnerability detection, there is a need for an end-to-end approach that seamlessly incorporates scalable program analysis techniques, modeling of security threats, and complex (string) constraints solving techniques.

5.3 Approach

Our approach is outlined in Figure 5.1, where dotted rounded rectangles correspond to global inputs/outputs, solid rounded rectangles correspond to intermediate inputs/outputs, solid rectangles correspond to operations, and dashed rectangles correspond to macro-steps.

The approach takes as input the bytecode of a Web application written in Java, a catalogue of vulnerabilities, and a list of threat models; it yields a vulnerability report. The catalogue of vulnerabilities contains a characterization of the criteria for identifying *input sources* and *sinks* related to specific vulnerabilities. The vulnerability report lists the vulnerabilities found in the Web application, and for each of them indicates the type and the corresponding sink. Our approach is composed of two macro-steps:

1. *Security slicing & Attack conditions generation.* This step first performs *security slicing* which is explained in Chapter 3. For each sink it computes the path condition leading to it and the associated context information. The path condition and the context information of each sink are then used, together with the list of threat models, to generate *attack conditions*, i.e., conditions that could trigger a security attack over a security slice. This step is explained in Section 5.3.1.

2. *Constraint solving.* This step takes as input the attack conditions generated in the previous macro-step, in the form of a constraint. This constraint is first pre-processed to simplify

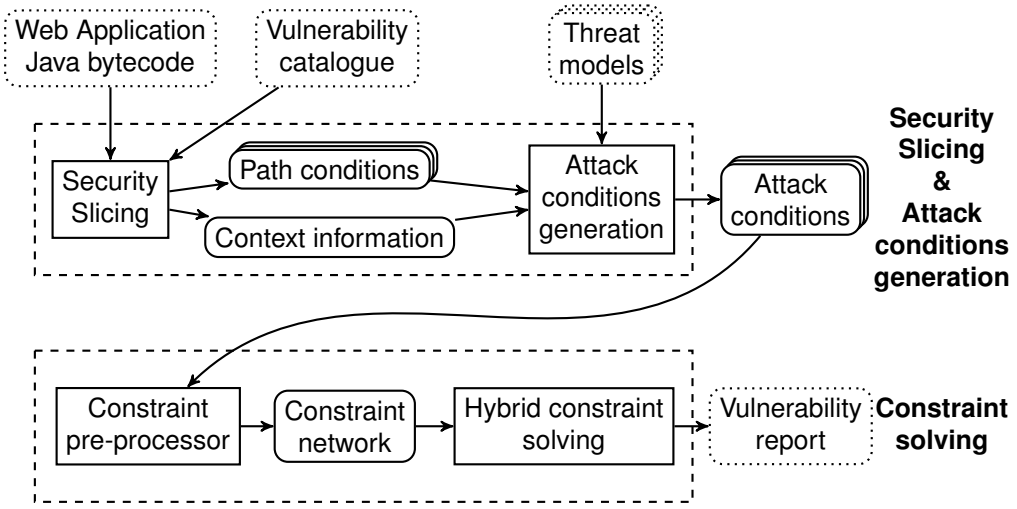


Figure 5.1: Overview of the integrated approach.

it through the *constraint pre-processing* step (detailed in Section 5.3.2). The resulting constraint, represented as a constraint network, is then given as input to a *hybrid constraint solver*, which orchestrates a constraint solving procedure for string and integer constraints with our search-based constraint solving procedure (see Chapter 4); more details of this step are presented in Section 5.3.3. The results yielded by the hybrid constraint solver are used to create the vulnerability report.

5.3.1 Security Slicing and Attack Conditions Generation

In Chapter 3 we have explained security slicing, which is a technique that extracts all the program statements required for auditing the security of a given sink. A security slice is a concise and minimal sequence of program statements necessary to determine the vulnerability of a sink.

To identify sources and sinks in the Web application under test, security slicing relies on a vulnerability catalogue, i.e., a predefined set of sink and source signatures, which can be easily extended by adding new signatures. We preconfigured a default vulnerability catalogue that contains a rich set of commonly used API signatures.

Security slicing performs symbolic execution on each path in a security slice to extract the path condition(s) characterizing the path. However, from a security auditing standpoint it is also necessary to understand the *context* of a sink, i.e., *how* the input data is used in a sink. Examples of possible contexts are the content or an attribute of an HTML tag, as well as a quoted value of an SQL query. This information can be computed through *context analysis* (see Section 3.2.2.2) which identifies the context (within a sink) in which the data of an input source is used. More specifically, context analysis traces the values of the variables used in the sink along the path, to reconstruct the query (e.g., SQL/XPath/LDAP query) or the document part (e.g., HTML/XML fragment) that is being generated at the sink. Context information is computed by matching the reconstructed query or document part against some predefined patterns (shown in the "Context" column of Table 2.1).

We give a bird's eye view of attack condition generation through the running example

in Listing 3.1. Besides the XPath vulnerability that we used as an example in Section 4.2, the program is also vulnerable to XSS. The sink at Line 30 is vulnerable to XSS because of the inadequate sanitization procedure applied to variable `sid`, which contains a user input. More specifically, it is sanitized by applying a custom function `customSanit` (Line 27), which removes the meta-characters <, >, and / from the input string variable `sid`. string replacement operation. However, in this operation variable `sid` is used in two different contexts: as an URL parameter value (`...com?id=sid`) and as the content of an HTML element `<a href>...sid</a>`. The sanitization procedure used is appropriate for the second context since it prevents the injection of additional HTML tags like `<script>`. However, it is not appropriate for the first one, which would have required URL encoding (also called percent-encoding).

Figure 3.4a shows the corresponding security slice for the (XSS) sink at Line 30 in Listing 3.1. The security slice shown in Figure 3.4a contains only two paths leading to the sink. The first path is characterized by path condition *PC1*, which corresponds to the path that follows the true branch of the selection statement at Line 18 in Figure 3.4a and leads to the execution of the sink; this path condition is:

$$\begin{aligned} PC1 \equiv\ & \texttt{Integer.parseInt(}MAX\texttt{)} > \texttt{20} \wedge \\ & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \\ & \wedge\ SID\texttt{.length()} > \texttt{20} \end{aligned}$$

The second path is characterized by path condition *PC2*, which corresponds to the path that follows the false branch of the selection statement at Line 18 in Figure 3.4a and leads to the execution of the sink; this path condition is:

$$\begin{aligned} PC2 \equiv\ & \neg(\texttt{Integer.parseInt(}MAX\texttt{)} > \texttt{20}) \wedge \\ & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \\ & \wedge\ SID\texttt{.length()} > \texttt{Integer.parseInt(}MAX\texttt{)} \end{aligned}$$

For both paths, our context analysis procedure (see Chapter 3) identifies the following two contexts: *CTX1*, in which the symbolic expression `customSanit(`SID`)` is used as a URL parameter value in an XSS sink; *CTX2*, in which the symbolic expression `customSanit(`SID`)` is used as an element content in an XSS sink. Note that the symbolic expression `customSanit(`SID`)` represents the values of variable sid used at the sink.

The output of security slicing—path conditions and context information—is used to generate attack conditions, represented as constraints. A new constraint is generated for each context identified for each path, based on the threat model characterizing the security threat in that specific context. The Attack Condition Generation (ACG) process follows three steps:

ACG1 Since different contexts require different threat models, the procedure determines the appropriate threat model for a given context by looking up the list of threat models provided as input. The identification of the threat model is a fully automated procedure that matches the context returned by security slicing with one of the entries in the threat models list. The predefined version of this list is presented in Section 2.1.3; though not showed in Table 2.1, the predefined list also contains catch-all entries[1] for

[1]The catch-all entries for threat models are more generic and might lead to false positives in terms of vulnerability detection.

each type of vulnerability, which are used as fallback mechanism when there is no context matching pattern. Furthermore, the structure of the list guarantees that there is always only one applicable threat model for a given context. For example, the threat model for context *CTX1* is #7 in Table 2.1; likewise, the threat model for *CTX2* is #1.

ACG2 In the constraint corresponding to each identified threat model, the symbol $input$ is replaced with the actual symbolic expression of the input. This results in a constraint that checks if an input used at the sink contains a security attack. For example, the constraint $input$`.matches(".*['\"=<>/,;+-&\*\[\] ].*")` —corresponding to threat model #7 (see Table 2.1)— results in the following constraint *ATTK1*, related to context *CTX1*:

$$ATTK1 \equiv \texttt{customSanit(}SID\texttt{)} \texttt{.matches(".*['\"=<>/,;+-&*\[\]].*")}$$

Likewise, the constraint *ATTK2* related to *CTX2* is:

$$ATTK2 \equiv \texttt{customSanit(}SID\texttt{).matches(".*[<>/].*")}$$

ACG3 For each constraint generated in the previous step and a given path condition, the attack conditions are generated by simply conjoining the path condition with the constraint. For example, the attack condition *SEC1* is the constraint conjoining *PC1* and *ATTK1*:

$$\begin{aligned} SEC1 \equiv\ & \texttt{Integer.parseInt(}MAX\texttt{)} > \texttt{20} \wedge \\ & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \wedge \\ & SID\texttt{.length()} > \texttt{20} \wedge \\ & \texttt{customSanit(}SID\texttt{)} \\ & \texttt{.matches(".*['\"=<>/,;+-&*\[\]].*")} \end{aligned}$$

Likewise, attack condition *SEC4* conjoins *PC2* and *ATTK2*:

$$\begin{aligned} SEC4 \equiv\ & \neg(\texttt{Integer.parseInt(}MAX\texttt{)} > \texttt{20}) \wedge \\ & OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \wedge \\ & SID\texttt{.length()} > \texttt{Integer.parseInt(}MAX\texttt{)} \wedge \\ & \texttt{customSanit(}SID\texttt{).matches(".*[<>/].*")} \end{aligned}$$

Similar attack conditions (omitted for space reasons) are computed by conjoining *PC1* and *ATTK2*, as well as *PC2* and *ATTK1*.

More details on security slicing and context analysis are available in Section 3.2.2.2.

5.3.2 Constraint Preprocessing

Constraints corresponding to attack conditions generated from the previous steps are represented as constraint networks, which are a common representation of instances of a constraint satisfaction problem [70].

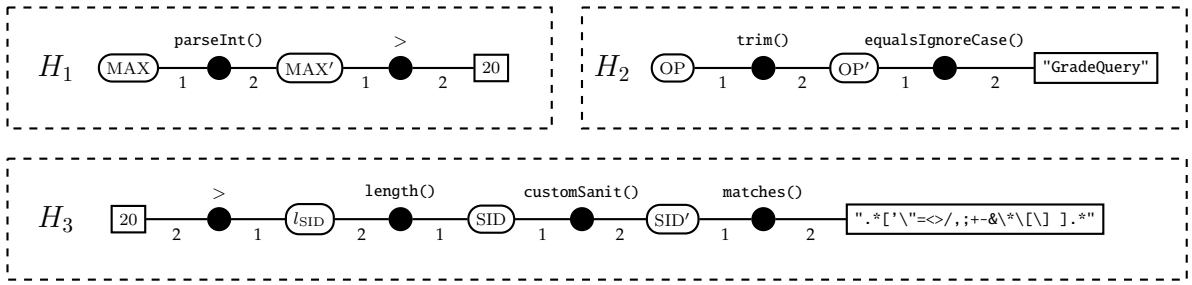


Figure 5.2: Constraint network equivalent to the attack condition *SEC1*

We build the hypergraph of the constraint network representing an attack condition similarly to how string graphs are constructed in [103]. The nodes of the graphs are either constant values or string/integer variables appearing in the constraints of the network. Each operation (e.g., method calls like `trim` or comparison operators like $>$) in an attack condition corresponds to exactly one hyperedge in the constraint network graph. Notice that boolean operators are represented as hyperedges labeled with the operator itself. Transformational operations (which return a value of type different from boolean) require to add a node to the graph, representing an auxiliary variable that corresponds to the result of the operation; the transformational operation is then added as a hyperedge that connects the nodes of the initial constants/variables and the new auxiliary variable. The latter is denoted with a name ending with a prime ($'$) symbol, except when the transformational operation is `length`, in which case we use a name of the form l_{VAR}, where VAR is the original variable.

For example, the constraint network corresponding to attack condition *SEC1* is shown in Figure 5.2. Rounded nodes represent variables, squared nodes represent constant values, hyperedges are denoted by lines that meet at a black dot. A hyperedge is labeled with the name of the corresponding operation and with numbers that indicate the role of its component nodes in the corresponding operation (i.e., order of function arguments, return variable). For instance, the node labeled MAX' corresponds to the auxiliary variable resulting from the application of the transformational operation `parseInt` to variable MAX; the node labeled l_{SID} is the auxiliary variable that represents the length of variable SID. Notice that, to keep the figures readable, we omit the representation of hyperedges labeled with a boolean AND.

Once a constraint network is constructed, we preprocess it to apply some rules that simplify the solving procedure; these rules are captured by two preprocessing procedures:

1. *derived constraint generation* and

2. *constraint refinement*,

executed in this order and applied through a work-list algorithm. Both procedures have been already described in [103]; here we propose new derived constraints and new rules to deal with additional operations. We remark that these two procedures preserve the integrity of the original constraint; in other words, the original and modified versions of the constraint network are equi-satisfiable.

Note that when some of the variables or the constraints of a constraint network are independent, the underlying hypergraph is disconnected; in such a case, we apply the preprocessing

Table 5.1: String/mixed operations and their corresponding derived constraints (X and Y are string variables, X' is an auxiliary string variable, i and j are integer variables, I' is an auxiliary integer variable, l_T represents the length of string T).

String/mixed operation	Derived constraint(s)	Source
X.contains(Y)	$l_X \geq l_Y$	[103]
X.startsWith(Y)	$l_X \geq l_Y$	[103]
X.endsWith(Y)	$l_X \geq l_Y$	[103]
X.isEmpty()	$l_X = 0$	⋆
X.concat(Y)	$l_{X'} = l_X + l_Y$	[41, 103]
$X + Y$		[41, 103]
X.append(Y)		⋆
X.equals(Y)	$l_X = l_Y$	[41, 103]
X.equalsIgnoreCase(Y)		⋆
X.contentEquals(Y)		⋆
String.copyValueOf(X)	$l_X = l_{X'}$	⋆
valueOf(X)		⋆
X.toString()		⋆
X.trim()	$l_X \geq l_{X'}$	[41, 103]
X.length()	$l_X \geq 0$	[41, 103]
X.indexOf(Y)	$(I' \geq 0 \rightarrow \neg(X.\texttt{substring}(0, I').\texttt{contains}(Y)) \wedge X.\texttt{substring}(I', l_X).\texttt{startsWith}(Y) \wedge l_X \geq l_Y) \wedge (I' < 0 \rightarrow \neg(X.\texttt{contains}(Y)))$	⋆
X.lastIndexOf(Y)	$(I' \geq 0 \rightarrow \neg(X.\texttt{substring}(I' + 1, l_X).\texttt{contains}(Y)) \wedge X.\texttt{substring}(I', l_X).\texttt{startsWith}(Y) \wedge l_X \geq l_Y) \wedge (I' < 0 \rightarrow \neg(X.\texttt{contains}(Y)))$	⋆
X.charAt(i)	$l_{X'} = 1 \wedge X.\texttt{contains}(X')$	⋆
X.toLowerCase()	$l_X = l_{X'}$	⋆
X.toUpperCase()	$l_X = l_{X'}$	⋆
X.isEmpty()	$l_X = 0$	⋆
X.substring(i)	$0 \leq i < l_X \wedge l_{X'} = l_X - i \wedge X.\texttt{contains}(X')$	⋆
X.substring(i,j)	$0 \leq i < l_X \wedge i \leq j \leq l_X \wedge l_{X'} = j - i \wedge X.\texttt{contains}(X')$	⋆

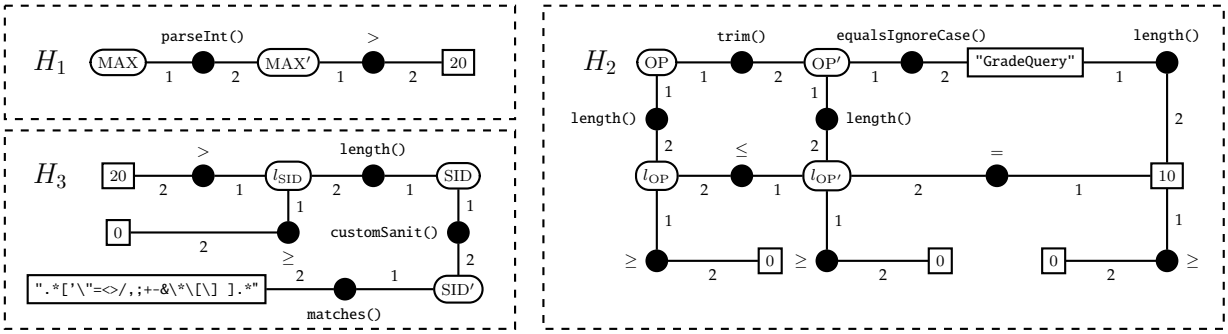


Figure 5.3: The constraint network in Figure 5.2 augmented with the derived constraints.

procedures to each of the maximal connected components (i.e., hypersubgraphs) of the hypergraph. For example, in Figure 5.2, there are three hypersubgraphs, enclosed in dashed rectangles and denoted as H_1, H_2, and H_3.

Table 5.2: Patterns for the refinement of integer constraints used in rule 3 of Table 5.3.

pattern	equivalent constraint
$X \geq Y \wedge X > Y$	$X > Y$
$X \geq Y \wedge X = Y$	$X = Y$
$X \geq Y \wedge X \neq Y$	$X > Y$
$X \leq Y \wedge X < Y$	$X < Y$
$X \leq Y \wedge X = Y$	$X = Y$
$X \leq Y \wedge X \neq Y$	$X < Y$
$X \leq Y \wedge X \geq Y$	$X = Y$

5.3.2.1 Derived Constraint Generation

For some operations, additional constraints are derived to reduce the input domain or to solve the constraints more efficiently. For example, given the constraint X`.contains(`Y`)`, one can generate the derived constraint $l_X \geq l_Y$ on the length of X and Y. The addition of these derived constraints reduces the size of the variable domains and may lead to unsatisfiability results faster, since some of the new derived constraints may be easier to solve (e.g., because they use a smaller number of variables).

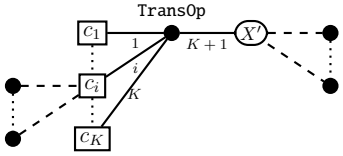
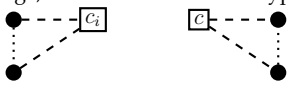
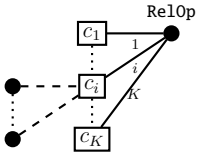
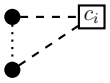
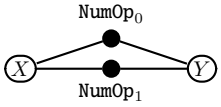
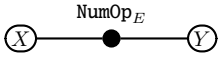
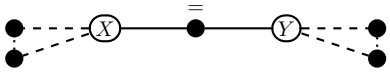
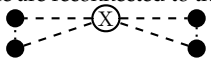
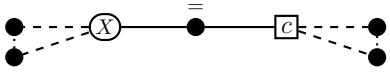
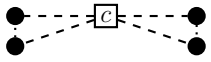
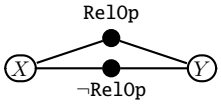
Table 5.1 shows the derived constraints corresponding to string/mixed operations; the derived constraints marked with a star are introduced for the first time in this work, while the others have been taken from [41, 103]. Notice that some operations in Table 5.1 may return a specific value to indicate an error: for example, `indexOf` returns a negative value when the search string is not found; we model this semantics using logical implications.

The derived constraints are then included in the constraint network accordingly. For example, Figure 5.3 shows the constraint network obtained after generating the derived constraints for the network in Figure 5.2; the new constraints are derived from the `trim`, `equalsIgnoreCase`, and `length` operations.

5.3.2.2 Constraint Refinement

In this step, some rules are used to simplify the constraint network and to detect trivially inconsistent constraints, to avoid unnecessary and expensive constraint solving. For example, if there is a constraint of the form X`.equals(`Y`)`, the hyperedge corresponding to the equality constraint and one of the nodes (either X or Y) can be removed, and its connected hyperedges can be redirected to the remaining node.

Table 5.3: Hyperedges and their corresponding refinement rules. (X, Y, X_1, X_K, Y_1, Y_K are variables, X' and Y' are auxiliary variables, $c_1, \ldots c_K, 1 \leq i \leq K$ and c are constants)

ID	Hyperedge(s)	Refinement Rule	Source
1	`TransOp`$(c_1, \ldots, c_K)$ where `TransOp` is any transformational operation that involves only constants. 	The operation `TransOp` is executed to determine its result c. The node corresponding to the auxiliary variable is replaced with the constant c. The hyperedge and the component nodes c_i, $1 \leq i \leq K$, not connected to any other hyperedge, are removed from the hypergraph. 	[87]
2	`RelOp`$(c_1, \ldots, c_K)$, where `RelOp` is any relational or boolean operation that involves only constants. 	If the operation `RelOp` returns `true`, the hyperedge and the component nodes c_i, $1 \leq i \leq K$, not connected to any other hyperedge, are removed from the hypergraph. Otherwise, constraint unsatisfiability is detected. (The figure below corresponds to the case in which `RelOp` returns `true`). 	[87]
3	A pair of hyperedges corresponding to a pair of integer constraints that matches one of the patterns shown in the left column of Table 5.2. 	The two hyperedges are replaced by the hyperedge representing the equivalent constraint $\texttt{NumOp}_E$ indicated in the right column of Table 5.2. 	[44]
4	X`.equals(`Y`)`, $X = Y$. 	The hyperedge and one of the component nodes (X or Y) are removed. The hyperedges that were connected to the removed node are reconnected to the other. 	[103]
5	X`.equals(`c`)`, $X = c$. 	Same as above, except that the variable X, not the constant c, is removed. 	[103]
6	A pair of hyperedges corresponding to inconsistent constraints between the same nodes, e.g., X`.contains(`Y`)` and $\neg X$`.contains(`Y`)`. 	Constraint unsatisfiability is detected.	[103]

continues on next page

Table 5.3 – *continues from previous page*

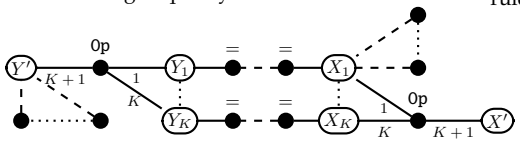
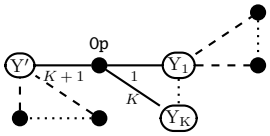
7	A pair of hyperedges that are labeled with the same operation and whose parameters are connected through equality constraints.		The two hyperedges are merged into a single hyperedge and the component nodes are processed according to rules 4 and 5. 	[81]

Table 5.3 shows the constraint refinement rules for specific types of hyperedges (or pair of hyperedges), as well as a pictorial representation of them; the rules marked with a star are proposed for the first time in this work, while the others have been taken from [103, 81].

Rule 1 corresponds to the evaluation of a transformational operation involving only constants. The operation is actually executed to determine its result; the node corresponding to the auxiliary variable is replaced by a single constant node corresponding to the computed result. The hyperedge labeled with the transformational operation is removed from the hypergraph, and also its associated constant nodes, if they are not connected to any other hyperedge.

Rule 2 is similar to the previous; it corresponds to the case in which a hyperedge is labeled with a relational or boolean operation involving only constants. The operation is evaluated; either it evaluates to true, and thus the hyperedge is removed from the hypergraph, or to false, and thus it determines the unsatisfiability of entire constraint network.

Rule 3 corresponds to the case in which there is a pair of hyperedges representing integer constraints, which matches one of the patterns shown in the left column of Table 5.2. This pair is replaced by a hyperedge representing the equivalent constraint, as indicated in the right side of Table 5.2. This table is based on well-known equivalences between numeric constraints. For example, the pair of hyperedges equivalent to a constraint of the form $X \geq Y \wedge X = Y$, is replaced by one hyperedge corresponding to the constraint $X = Y$.

Rules 4 and 5 are applicable to a hyperedge that corresponds to an equality constraint between two variables and, respectively, to an equality constraint between a variable and a constant. In both cases, the hyperedge corresponding to the equality constraint is removed, as well as one of the component nodes (the node corresponding to the variable in rule 5); the hyperedges that were connected to the removed node are reconnected to the other.

Rule 6 captures pairs of hyperdeges corresponding to inconsistent constraints (imposed on the same variables) of the form $\texttt{RelOp}(v_1, \ldots, v_K) \wedge \neg\texttt{RelOp}(v_1, \ldots, v_K)$; this rule determines the unsatisfiability of the entire constraint network.

Rule 7 is applicable to pairs of hyperedges that are labeled with the same operation and whose parameters are connected through equality constraints; this means that the two hyperedges are semantically equivalent. They are merged into a single hyperedge and the component nodes are processed according to rules 4 and 5.

Rules 4–7 use the theory of Equality of Uninterpreted Functions (EUF) [81], a widely used theory in constraint solving, to identify and merge semantically equivalent nodes and hyperedges.

As an example, the constraint network in Figure 5.3 is refined into the constraint network shown in Figure 5.4. More specifically, since the constraint $10 \geq 0$ in H_2 trivially evaluates to true, according to rule 2 the hyperedge labeled with $\geq$ and the constant node 0 are removed

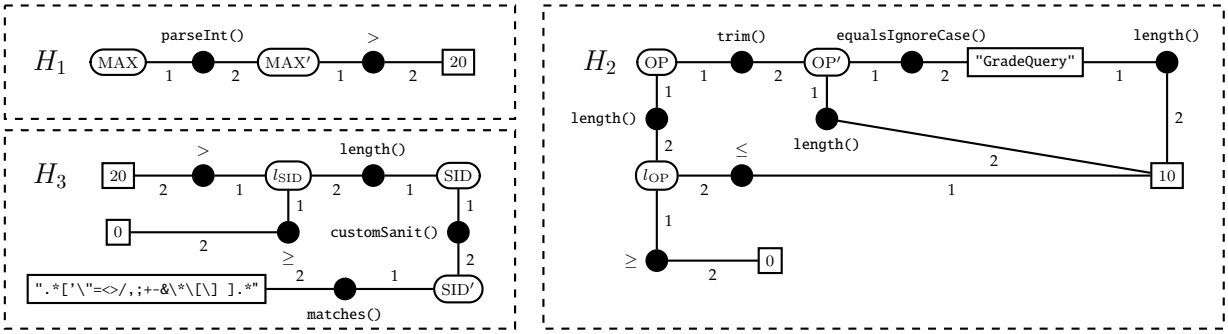


Figure 5.4: Constraint network resulting from the application of the constraint refinement rules to the network in Figure 5.3.

from the network; also, according to rule 5, in H_2 the hyperedge corresponding to constraint $l_{OP'} = 10$ as well as the variable node for $l_{OP'}$ are removed and the hyperedges that were connected to the latter are now connected to the node for constant 10; since one of the resulting hyperedges corresponds to the constraint $10 \geq 0$, rule 2 can be applied again as above, to remove the hyperedge.

5.3.3 Hybrid Constraint Solving

The constraint network resulting from the previous pre-processing step is then solved by our *hybrid* constraint solver. Our solver is hybrid because it orchestrates a constraint solving procedure for string/mixed and integer constraints with our search-based constraint solving procedure presented in Chapter 4.

The idea behind this hybrid solving strategy is to solve a constraint network through a two-stage process: in the first stage, our solving procedure solves all the constraints with supported operations, providing a unified treatment for string and integer constraints. In the second stage, we use our search-driven solving procedure based on the Ant Colony Optimization meta-heuristic to solve the remaining constraints which contain unsupported operations. The solver in the first stage is used to reduce (possibly in a significant way) the search space, i.e., the domains of the string and integer variables, for the search-driven solving procedure; hence, it makes the search in the second stage more scalable and effective.

The pseudocode of our string constraint solving algorithm CSTRSOLVE is shown in Algorithm 5.6. It takes as input a hypergraph H corresponding to the constraint network to solve and returns whether it is satisfiable, unsatisfiable, or whether it timed out; when it returns satisfiable, it also returns the set of solutions found, which are used to build the vulnerability report.

First, it computes (line 4) the set HS of connected hypersubgraphs of H by means of function GETMCHYPERSUBGRAPHS. Then, it iterates through all the elements $H_i \in HS$ to perform the following steps (lines 5–13).

Function SOLVESUPPORTEDOPS (line 6) solves the constraints in H_i containing supported operations and returns the set Sol, which contains the solutions for all the string and integer variables in H_i; the details of SOLVESUPPORTEDOPS are presented in Section 5.3.3.1.

Subsequently, if H_i contains any constraints containing unsupported operations, the algorithm invokes the SEARCHSOLVE function, which implements a meta-heuristic search algorithm

```
1: function CSTRSOLVE(Hypergraph H)
2:     Boolean solved ← false
3:     Set of Solution Sol ← ∅
4:     Set of Hypergraph HS ← GETMCHYPERSUBGRAPHS(H)
5:     for all H_i ∈ HS do
6:         Sol ← SOLVESUPPORTEOPS(H_i, Sol)
7:         if CONTAINSUNSUPPORTEDOPS(H_i) then
8:             ⟨solved, Sol⟩ ← SEARCHSOLVE(H_i, Sol)
9:             if ¬solved then
10:                return TIMEOUT
11:            end if
12:        end if
13:    end for
14:    return ⟨SAT, Sol⟩
15: end function
16: function SOLVESUPPORTEDOPS(Hypergraph H, Set of Solution Sol)
17:    Sol ← INITIALIZE(Sol)
18:    Set of Hyperedge Worg ← W ← GETSUPPORTEDEDGES(H)
19:    repeat
20:        Hyperedge e ← SELECTEDGE(W)
21:        Set of Solution newSol ← APPLYRECIPE(e,Sol)
22:        Sol ← UPDATE(newSol, Sol)
23:        W ← UPDATEWORKLIST(newSol,Worg,W)
24:        W ← W \ {e}
25:    until ISEMPTY(W)
26:    return Sol
27: end function
```

Algorithm 5.6: Hybrid Constraint solving algorithm.

(detailed in Section 4.3.2). SEARCHSOLVE tries to find a solution for the constraints which could not be solved by SOLVESUPPORTEDOPS due to the presence of unsupported string or integer operations and, thus, provides a general mechanism for solving them. SEARCHSOLVE returns a flag *solved* and an updated set of solutions *Sol* (line 8). If the flag *solved* is true, it means that the constraints in H_i have been solved and the algorithm can proceed to process the hypersubgraph H_{i+1}; otherwise, it means that the SEARCHSOLVE function timed out and, thus, the algorithm returns TIMEOUT, terminating the entire constraint solving procedure. A time-out can indicate either that a solution exists for the constraint but the solver could not find it, or that the constraint is actually unsatisfiable. The algorithm returns SAT and the set of solutions *Sol* (Line 14) only when the loop over H has been completely executed, meaning that the attack condition corresponding to the constraint network given as input is satisfiable.

Representation of the solutions. The search-based constraint solving step requires every variable domain to be represented in the form of a solution automaton. A solution automaton is an FSM accepting the language determined by the constraints imposed on the variable; in our case, a solution automaton for a variable accepts only the language corresponding to the set of values (for the variable) that satisfy *the constraints the solver has solved so far.* Hence, the set *Sol* computed at Line 6 and Line 8 contains solution automata. Notice that we provide a unified treatment of integer and string constraints by converting integer ranges into their automaton representation. For example, the solution automaton for a string variable s involved in a constraint `s.matches("abc*")` would be `"abc*"`; the solution automaton for an integer variable i involved in the constraint $i > 2$ is `[3-9] | [1-9] [0-9]`$^+$, corresponding to the range $[3, \infty)$. For a variable involved only in constraints with unsupported operations, its corresponding solution automaton is the default one, accepting any value (i.e., the automaton accepting the regular language `0` $|\ (-?\ $`[1-9] [0-9]`$^*)$ for an integer variable and .* for a string variable).

5.3.3.1 Solving supported Operations

Our solving procedure leverages automata-based solving for solving string/mixed constraints, and linear interval arithmetic for solving integer constraints[2]. In both cases, constraint solving rules are expressed using *recipes* that model the semantics of the operations.

We specifically use automata-based solving and linear interval arithmetic (vs. bit-vector-based or word-based methods) because both methods return, when successful, a solution range for each variable occurring in the constraints they could solve, based on the operations they support.

The pseudocode of our SOLVESUPPORTEDOPS algorithm for solving constraints with supported operations is shown in Algorithm 5.6; it takes as input a hypergraph H and set of solutions Sol. First, the algorithm initializes each variable in set Sol with a default solution automaton, which is $.^*$ for string variables and $\texttt{0}|\,(-?\,[\texttt{1-9}]\,[\texttt{0-9}]^*)$ for integer variables (Line 17), and extracts from H the set of hyperedges W (and its copy $Worg$) labeled with supported operations (Line 18). Next, the constraints are solved using a worklist procedure (Lines 19–25). First, a hyperedge e is selected from W (Line 20). Then the APPLYRECIPE procedure (Line 21) processes the domains of the variables in the component nodes of e according to the recipe that models the semantics of the operation labeling e. We provide recipes (see Table 5.4 and Table 5.5) for both string and integer variables: string/mixed constraints are solved by means of automaton-operations (such as union, intersection, concatenation), and integer constraints are solved through linear interval arithmetic (where integer domains are represented by intervals).

As a result, SOLVESUPPORTEDOPS returns the set $newSol$ (Line 21), in which the solution automata of the variables in the component nodes of e accept the languages corresponding to the sets of values that satisfy the string/mixed or integer constraints represented by e. The set Sol is then updated with the solutions in $newSol$ (Line 22). If the solution automaton of a variable v is affected by this update, it needs to be checked for satisfiability with other constraints that involve v; in this case, the function UPDATEWORKLIST puts back into W all the previously-removed hyperedges (determined from the original list $Worg$) which are connected to the node for variable v (Line 23). Finally, the algorithm removes the hyperedge e from the set W. This worklist procedure is repeated until the set W is empty, i.e., until all the constraints with supported operations have been successfully solved. Notice that when a recipe results in an automaton that accepts the empty language, meaning that there is no solution to satisfy a constraint, the entire constraint solving procedure terminates internally and returns UNSAT.

The above worklist procedure implements the fixpoint iteration based on an arc-consistency algorithm [68]. A hyperedge is added back to the worklist (through the UPDATEWORKLIST operation) only if the domains of its variables could be further restricted through the application of a recipe. Since all the recipes in Table 5.4 and Table 5.5 are of the form $A := A \cap B$, i.e., they only restrict the domains of the involved variables, the worklist procedure is guaranteed to converge and to terminate with two possible results: (1) one variable domain becomes empty or (2) the constraint network stabilizes because none of the variable domains can be further restricted.

In the following subsections, we discuss the constraints that are supported by our automata-based+linear interval arithmetic solver and illustrate the worklist procedure with an application

[2] Integer constraints that cannot be solved with linear interval arithmetic are treated as constraints with unsupported operations.

to the running example.

Supported String and Mixed Constraints Our automata-based solver supports many operations from the `java.lang.String` class, except methods like `format` and `hashCode` that cannot be expressed in terms of FSMs. The solver also supports the `matches`, `parseInt`, `parseLong`, `toString`, `valueOf`, `append`, and `length` operations from other Java classes that are commonly found in Java applications [102]. In addition, the solver supports 16 input sanitization operations from the Apache Commons Lang 3 [8] and OWASP [94] standard security libraries; we remark that in the context of security analysis it is important to support such operations for achieving precise and efficient analysis.

Table 5.4 shows a subset of operations supported by our automata-based solver and the corresponding recipes[3]; the operations marked with a star are proposed for the first time in this work, while the others have been taken from [103, 40]. In the table we adopt the following notation: M_X denotes the automaton accepting the language $L_X = \{X\}$, containing only the string X; M_ε denotes the automaton that accepts the empty string; $.^i$ denotes the automaton that accepts i number of any characters; * denotes the Kleene-star operator; $\neg$ denotes the complement operation; $\oplus$ denotes concatenation; $M_X \cap{:=} M_Y$ is a short-hand for $M_X := M_X \cap M_Y$; $[c_0 c_1 \ldots c_n]$ is a short-hand for $c_0 \cup c_1 \cup \ldots c_n$ where c stands for a character or a character range. In addition, we use a number of auxiliary operations that work on automata: $Prefix(M)$ and $Suffix(M)$ return the prefixes and suffixes of the words accepted by M, respectively; $Substring(M)$ returns an automaton that accepts the substrings of the words accepted by M; $Substring(M, i)$ returns an automaton that accepts the substrings starting from the index i of the words accepted by M; $Trim(M)$ returns an automaton that accepts all words of M without leading and trailing blanks; $CharAt(M, i)$ returns an automaton that accepts the characters at the index i of the words accepted by M; $LowerCase(M)$ and $UpperCase(M)$ return an automaton that accepts the lowercase (respectively, uppercase) words of the words accepted by M; $Replace(M_X, c_1, c_2, M_{X'})$, $ReplaceFirst(M_X, c_1, c_2, M_{X'})$, and $ReplaceAll(M_X, c_1, c_2, M_{X'})$ are functions defined in [27, 40] modeling the homonymous replacement string operations, which return a tuple of solution automata M_X and $M_{X'}$, where $M_{X'}$ accepts the words resulting from the semantics of the corresponding replacement operation; $ApacheSanitizeHTML3(M_X, M_{X'})$ returns a tuple of solution automata M_X and $M_{X'}$ where $M_{X'}$ accepts the words resulting after replacing all occurrences of HTML meta-characters in the words accepted by M_X with their corresponding escape characters. The latter models the semantics of a standard sanitization function from Apache [8], through a series of *Replace* functions for ⟨meta-character, escape character⟩ pairs; we model the remaining 15 standard sanitization functions we support in a similar way.

Table 5.4: Automata operations (recipes) corresponding to string/mixed operations. (X and Y are string variables, X' is an auxiliary variable, c, c_1 and c_2 are string constants, i, i_1 and i_2 are numeric constants)

Operation	Recipe	Source
X`.charAt(`i`)`	$M_X \cap{:=} .^i \oplus M_{X'} \oplus .^*$ $M_{X'} \cap{:=} CharAt(M_X, i)$	[103]

continues on next page

[3]The full list of supported operation is available online [128].

Table 5.4 – *continues from previous page*

X.concat(Y) X.append(Y) X+Y	$M_X \cap \coloneqq \mathit{Prefix}(M_{X'})$ $M_Y \cap \coloneqq \mathit{Suffix}(M_{X'})$ $M_{X'} \cap \coloneqq M_X \oplus M_Y$	[103] [103] $\star$
X.contains(Y)	$M_X \cap \coloneqq .^* \oplus M_Y \oplus .^*$ $M_Y \cap \coloneqq \mathit{Substring}(M_X)$	[103]
copyValueOfX) valueOf(X) X.toString()	$M_X \cap \coloneqq M_{X'}$ $M_{X'} \cap \coloneqq M_X$	$\star$
X.endsWith(Y)	$M_X \cap \coloneqq .^* \oplus M_Y$ $M_Y \cap \coloneqq \mathit{Suffix}(M_X)$	[103]
X.equals(Y) X.contentEquals(Y)	$M_X \cap \coloneqq M_Y$ $M_Y \cap \coloneqq M_X$	[103]
X.equalsIgnoreCase(Y)	$M_X \cap \coloneqq \mathit{LowerCase}(M_Y) \cup \mathit{UpperCase}(M_Y)$ $M_Y \cap \coloneqq \mathit{LowerCase}(M_X) \cup \mathit{UpperCase}(M_X)$	$\star$
escapeHtml3(X)	$\langle M_X, M_{X'} \rangle \coloneqq \mathit{ApacheSanitizeHTML3}(M_X, M_{X'})$	$\star$
X.indexOf(Y)	$M_X \cap \coloneqq \{.^i \cap \neg M_Y\} \oplus M_Y \oplus .^*$ $M_Y \cap \coloneqq \mathit{Substring}(M_X, i)$	[103]
X.isEmpty()	$M_X \cap \coloneqq M_\varepsilon$	$\star$
X.lastIndexOf(Y)	$M_X \cap \coloneqq (.^* \oplus M_Y \oplus .^*) \cap \{.^{i+1} \oplus (.^* \cap \neg M_Y)\}$ $M_Y \cap \coloneqq \mathit{Substring}(M_X, i)$	[103]
X.length()	$M_X \cap \coloneqq .\{i_1, i_2\}$ where i_1 and i_2 are equal to the lower limit and the upper limit, respectively, of the length of X. If the upper limit is ∞, i_2 is removed from the expression, i.e., $.\{i_1,\}$. If X has a fixed length, the expression is $.\{i_1\}$.	$\star$
X.matches(c)	$M_X \cap \coloneqq M_c$	[40]
parseInt(X) parseLong(X)	$M_X \cap \coloneqq M_{X'} \cap (\texttt{0} \cup (\texttt{-}\{0,1\}[\texttt{1-9}][\texttt{0-9}]^*))$	$\star$
X.replace(c_1, c_2)	$\langle M_X, M_{X'} \rangle \coloneqq \mathit{Replace}(M_X, c_1, c_2, M_{X'})$	[40]
X.replaceAll(c_1, c_2)	$\langle M_X, M_{X'} \rangle \coloneqq \mathit{ReplaceAll}(M_X, c_1, c_2, M_{X'})$	[40]
X.replaceFirst(c_1, c_2)	$\langle M_X, M_{X'} \rangle \coloneqq \mathit{ReplaceFirst}(M_X, c_1, c_2, M_{X'})$	[40]
X.startsWith(Y)	$M_X \cap \coloneqq M_Y \oplus .^*$ $M_Y \cap \coloneqq \mathit{Prefix}(M_X)$	[103]
X.substring(i)	$M_X \cap \coloneqq .^{i-1} \oplus M_{X'}$ $M_{X'} \cap \coloneqq \mathit{Substring}(M_X, i)$	[103, 40]
X.substring(i_1, i_2)	$M_X \cap \coloneqq .^{i_1} \oplus M_{X'} \oplus .^*$ $M_{X'} \cap \coloneqq \mathit{Substring}(M_X, i_1, i_2)$	[103, 40]
X.toLowerCase()	$M_{X'} \cap \coloneqq \mathit{LowerCase}(M_X)$ $M_X \cap \coloneqq M_{X'} \cup \mathit{UpperCase}(M_{X'})$	$\star$
X.toUpperCase()	$M_{X'} \cap \coloneqq \mathit{UpperCase}(M_X)$ $M_X \cap \coloneqq M_{X'} \cup \mathit{LowerCase}(M_{X'})$	$\star$
X.trim()	$M_X \cap \coloneqq [\,]^* \oplus M_{X'} \oplus [\,]^*$ $M_{X'} \cap \coloneqq \mathit{Trim}(M_X)$	[103]

Supported Integer Constraints Integer constraints are solved by means of linear interval arithmetic. When successful, this method returns a range of solutions for each integer variable involved in the constraints.

Our solver supports basic operations of the form $I\ op\ K$, where I, K are integer variables

Table 5.5: Interval operations (recipes) corresponding to integer constraints (L, N and K are integer variables, I_V stands for the interval of variable V with $I_V \coloneqq [v_{\min}, v_{\max}]$ where $v_{\min}$ and $v_{\max}$ denote lower and upper bounds, respectively).

Operation	Recipe	Source
$K = N$	$I_K \cap \coloneqq I_N$ $I_N \cap \coloneqq I_K$	[86, 151]
$K > N$	$I_{>N} \coloneqq [n_{\min} + 1, \infty)$ is the interval that captures all numbers greater than $n_{\min}$ $I_{<K} \coloneqq (-\infty, k_{\max} - 1]$ is the interval that captures all numbers smaller than $k_{\max}$ $I_N \cap \coloneqq I_{<K}$ $I_K \cap \coloneqq I_{>N}$	[151]
$K \geq N$	$I_{\geq N} \coloneqq [n_{\min}, \infty)$ is the interval that captures all numbers greater than or equals to $n_{\min}$ $I_{\leq K} \coloneqq (-\infty, k_{\max}]$ is the interval that captures all numbers smaller than or equals to $k_{\max}$ $I_N \cap \coloneqq I_{\leq K}$ $I_K \cap \coloneqq I_{\geq N}$	[151]
$L = K + N$	$I_L \cap \coloneqq [k_{\min} + n_{\min}, k_{\max} + n_{\max}]$ $I_K \cap \coloneqq [l_{\min} - n_{\max}, l_{\max} - n_{\min}]$ $I_N \cap \coloneqq [l_{\min} - k_{\max}, l_{\max} - k_{\min}]$	[86, 151, 50]
$L = K - N$	$I_L \cap \coloneqq [k_{\min} - n_{\max}, k_{\max} - n_{\min}]$ $I_K \cap \coloneqq [l_{\min} + n_{\min}, l_{\max} + n_{\max}]$ $I_N \cap \coloneqq [k_{\min} - l_{\max}, k_{\max} - l_{\min}]$	[86, 151, 50]

and $op \in \{=, >, <, \geq, \leq, +, -\}$. From our experience with the test subjects in our experiments[4] this set of supported operations is enough, since most of the constraints encountered when analyzing the injection vulnerabilities of Web applications are linear integer constraints; more details are provided in Section 5.5.

Table 5.5 shows the recipes for various integer operators op; in this case, a recipe is a sequence of operations that are executed to compute a range of solutions for each integer variable according to the semantics of op. These recipes have been taken from the sources ([86, 151, 50]) indicated in the last column of Table 5.5. In the table, we adopt the following notation: I_V denotes the interval of variable V with v_{min} and v_{max} being its lower and upper bounds, respectively; $I_K \cap \coloneqq I_N$ is a short-hand for $I_K \coloneqq I_K \cap I_N$, i.e., the intersection of the two intervals I_K and I_N.

As discussed above, we convert integer ranges into their automaton representation. More specifically, for each integer variable V, we generate an automaton representing its interval I_V with $I_V \coloneqq [v_{\min}, v_{\max}]$, i.e., the automaton that accepts only the string representation of numbers within I_V. We convert intervals by intersecting the automata for $[v_{\min}, \infty)$ and $(-\infty, v_{\max}]$. For example, the integer interval $I_V = [3, 45]$ can be represented as $[\texttt{3-9}] \,|\, [\texttt{1-9}]\,[\texttt{0-9}]^+ \cap \texttt{0}| - [\texttt{1-9}]\,[\texttt{0-9}]^*\,|\texttt{4}\,[\texttt{0-5}] \,|\, [\texttt{1-3}]\,[\texttt{0-9}] \,|\, [\texttt{0-9}]$.

[4] All of the numeric constraints that appeared in our test subjects could be solved by means of the recipes listed in Table 5.5.

5.3.3.2 Application to the Running Example

We now show an example of the execution of SOLVESUPPORTEDOPS when solving the hyper-subgraph H_3 in Figure 5.4, which contains string/mixed, and integer constraints. The solution automata for the two string variables SID and SID' are initialized with $.^*$; the solution automaton for the integer variable l_{SID} is initialized with $\texttt{0}|\,(-?\,[\texttt{1-9}]\,[\texttt{0-9}]^*)$. First, the procedure GETSUPPORTEDEDGES returns W (further assigned also to W_{org}), which contains four hyperedges, labeled with $>$, $\geq$, `length` and `matches`.

Let us assume that in the first iteration of the worklist loop, function SELECTEDGE selects the hyperedge labeled with $\geq$. This hyperedge has two component nodes: the integer variable node l_{SID} and the constant node 0. The solution for l_{SID} is solved by applying the recipe given in row 4 of Table 5.5, which yields $l_{SID} \coloneqq [0, \infty)$. The hyperedge $\geq$ is then removed from the worklist W and the worklist loop will continue to the next iteration; let us assume that the hyperedge labeled with $>$ is selected in the latter. Similarly to the previous iteration, the *new* solution for l_{SID} is obtained by applying the recipe given in row 3 of Table 5.5, which yields $l_{SID} \coloneqq [21, \infty)$. Since this results in a change to the existing solution automaton of l_{SID} in Sol, the previously-removed edge $\geq$, which is connected to the variable node l_{SID}, is put back into the worklist W; the hyperedge $>$ is also removed from W. Let us assume that the hyperedge labeled with $\geq$ is selected in the next iteration; by applying the recipe given in row 4 of Table 5.5, the solution for l_{SID} is $l_{SID} \coloneqq [21, \infty)$ which is converted to the automaton $2\,[\texttt{1-9}]\,|\,[\texttt{3-9}]\,[\texttt{0-9}]^+$. Since this results in no change to any existing solution automaton in Sol, no previously-removed edge is put back into W; the hyperedge $\geq$ is then removed from W. Subsequently, let us assume that the hyperedge labeled with `matches` is selected. This hyperedge has two component nodes: the string variable node SID' and the constant node `".*['\"=<>/,;+-&\*\[\] ].*"`, which is equivalent to the regular expression $.^*[\texttt{'"=<>/,;+-\&*[]]}.^*$. The solution automaton $M_{SID'}$ for the variable SID' is computed according to the recipe for the operation `matches` in Table 5.4:

$$
\begin{aligned}
M_{SID'} \coloneqq & M_{SID'} \cap .^*[\texttt{'"=<>/,;+-\&*[]]}.^* \leftrightarrow \\
M_{SID'} \coloneqq & .^* \cap .^*[\texttt{'"=<>/,;+-\&*[]]}.^* \leftrightarrow \\
M_{SID'} \coloneqq & .^*[\texttt{'"=<>/,;+-\&*[]]}.^*
\end{aligned}
$$

This means that the language accepted by $M_{SID'}$ is now restricted to the values matching the regular expression $.^*[\texttt{'"=<>/,;+-\&*[]]}.^*$. This results in a change to the existing solution automaton of SID' in Sol (recall that it was initialized with $.^*$). The procedure is supposed to put back any previously-removed hyperedge that is connected to node SID'; however, in this case there is no such edge. The hyperedge `matches` is then removed from W and the worklist loop continues to the next iteration, in which the remaining hyperedge `length` is selected. The solution automaton M_{SID} is then updated according to the recipe for operation `length` in Table 5.4, with $i_1 = 21, i_2 = \infty$:

$$
\begin{aligned}
M_{SID} \coloneqq & M_{SID} \cap .\{21,\} \leftrightarrow M_{SID} \coloneqq .^* \cap .\{21,\} \leftrightarrow \\
M_{SID} \coloneqq & .\{21,\}
\end{aligned}
$$

This means that the language accepted by M_{SID} is now restricted to the strings with a length greater than or equal to 21. This results in a change to the existing solution automaton of M_{SID} in Sol. Again, there is no previously-removed hyperedge connected to M_{SID} that has to be

[Iteration #100]

$T_{Best} = \{SID \coloneqq \texttt{aRXxQ1zCVmaetowbnZv0t}\}$
$f_{cstr_1} = 0$
$f_{cstr_2} = \psi(\texttt{aRXxQ1zCVmaetowbnZv0t}\ ,\ .^*[\texttt{'"=<>/,;+-\&*[]}\ ].^*) = 1$
$\hat{f}_{cstr_1} = 0;\ \hat{f}_{cstr_2} = 0.5$
$F_{Best} = avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}) = 0.25$

[Iteration #1000]

$T_{Best} = \{SID \coloneqq \texttt{\$Qaa.\&@erp!t'TmoopEn=}\}$
$f_{cstr_1} = 0$
$f_{cstr_2} = \psi(\texttt{\$Qaa.\&@erp!t'TmoopEn=}\ ,\ .^*[\texttt{'"=<>/,;+-\&*[]}\ ].^*) = 0$
$\hat{f}_{cstr_1} = 0;\ \hat{f}_{cstr_2} = 0$
$F_{Best} = avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}) = 0$

Figure 5.5: Results after 100 and 1000 iterations of the SEARCHSOLVE procedure.

put back in W. After this iteration, the worklist is empty and the algorithm returns the set $Sol = \{l_{SID}, M_{SID}, M_{SID'}\}$, where $l_{SID} \coloneqq 2\,[\texttt{1-9}]\,|\,[\texttt{3-9}]\,[\texttt{0-9}]^+$, $M_{SID} \coloneqq .\{21,\}$, and $M_{SID'} \coloneqq .^*[\texttt{'"=<>/,;+-\&*[]}\ ].^*$

Notice that the hyperedge labeled with `customSanit` is not in the worklist since it reflects an unsupported operation.

5.3.3.3 Solving unsupported Operations

The *searchSolve* method used in *JOACO* is the same as the one explained in Section 4.3.2. Here, we only show the application of SEARCHSOLVE to our example attack condition *SEC1*. We recall that hypersubgraph H_3 in Figure 5.4 was only partially solved through the application of function SOLVESUPPORTEDOPS in Section 5.3.3.2 since it contains an unsupported operation `customSanit`.

Procedure SEARCHSOLVE will attempt to find a value from the solution automata $M_{SID} \coloneqq .\{21,\}$ (determined by the automata-based solver) for the string variable *SID* that satisfies the two constraints $cstr_1 \equiv SID\texttt{.length()} > 20$; $cstr_2 \equiv \texttt{customSanit(}SID\texttt{).matches(".*['\"-=<>/,;+-\&*\[\]].*"))}$ in H_3. Figure 5.5 shows the best results (T_{Best}, f, $\hat{f}$, and F_{Best}) obtained after 100 and 1000 iterations; f_{cstr_1} is computed by evaluating the constraint $cstr_1$ with the value in T_{Best} and by using the Korel function; f_{cstr_2} is computed by evaluating the constraint $cstr_2$ with the value in T_{Best} and by using the equality cost function for regular expression matching. At iteration 1000, the search converges towards the desired solution for *SID*, which satisfies the constraints in H_3.

5.4 Implementation

We have implemented our approach in a tool called *JOACO* (available online [128]). It consists of two major components, the *security slicer* and the *constraint solver*; both are implemented in JAVA, comprising approximately 34 kLOC excluding library code, spaces and comments.

The *security slicer* is derived from *JoanAudit* (see Section 3.3) which is built on top of Wala [55] and Joana [47], which provide interprocedural program slicing and static analysis of paths in the slice, respectively. Given the bytecode of a Web program, the security slicer first extracts a security slice for each sink. It then explores the paths in the slice that lead to the sink in a depth-first manner, extracting the path conditions and the context information. The latter is used to generate the attack condition, by conjoining the path condition with the appropriate threat model. For scalability reasons, when encountering loops and recursive function calls, the slices iterates through them only once.

The *constraint solver* comprises three modules: constraint preprocessor, an automata-based and interval constraint solver and a search-based constraint solver. The constraint preprocessor makes use of the JGraphT library [90], a Java class library that provides mathematical graph-theory objects and algorithms, in order to generate a constraint network from the attack condition, as explained in Section 5.3.2. The constraint network is then passed to the constraint solver to prove the presence/absence of a vulnerability.

Our automata-based and interval constraint solver handles string and integer constraints with supported operations, as described in Section 5.3.3.1. It is built on top of JSA [27] and Sushi [40]. JSA models a set of Java string/mixed operations using finite state automata; Sushi adds supports for string replacement and regular expression replacement operations using finite state automaton and transducer operations. In this component, we also defined the recipes for additional string operations (see Table 5.4), such as the security APIs provided by two popular security libraries (OWASP [94] and Apache [8]). The search-driven constraint solver is invoked when a constraint contains unsupported operations, as described in Section 4.3.2.

5.5 Evaluation

In this section we report on the evaluation of *JOACO*, in terms of

1. vulnerability detection capability when analyzing the source code of a Web application;

2. capability of solving string constraints derived from potential vulnerabilities in realistic systems.

The first task corresponds to the normal usage scenario of *JOACO* for detecting vulnerabilities in Web applications. We also included the second usage scenario because many research efforts (see related work in Chapter 6) focus (only) on string constraint solving, as a means to enable vulnerability detection; indeed, in such a context, *JOACO* can be also used as a stand-alone string constraint solver (we call it *JOACO-CS* when used in this mode).

We assess the effectiveness of *JOACO* in performing these two tasks by answering the following research questions:

RQ1 What is the effectiveness of *JOACO* in detecting XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities and how does it compare with state-of-the-art vulnerability detection tools, including our previous work (see Chapter 3 and Chapter 4)? (Section 5.5.2.1)

RQ2 What is the effectiveness of *JOACO-CS* in solving string constraints characterizing potential vulnerabilities in representative and widely used systems and benchmarks and how does it compare with state-of-the-art, general-purpose string constraint solvers, including our previous work (see Chapter 4)? (Section 5.5.2.2)

RQ3 How does the constraint preprocessing described in Section 5.3.2 affect the execution time of *JOACO*? (Section 5.5.2.3)

5.5.1 Benchmarks and Evaluation Settings

We use five different benchmarks, obtained from different sources, to evaluate *JOACO*: *JOACO-Suite*, *Pisa-Suite*, *AppScan-Suite*, *Kausler-Suite*, and *Cashew-Suite*.

JOACO-Suite is our homegrown benchmark, composed of 11 open-source Java Web applications/services, with known XSS, XMLi, XPathi, LDAPi, and SQLi vulnerabilities. It is an extended version of the benchmark used in Section 4.5, enriched with two new applications: *Bodgeit* and *openmrs-module-legacyui*. *Bodgeit* [18] is a deliberately insecure Web application developed for the purpose of teaching security vulnerabilities in Web applications. *openmrs-module-legacyui* (shortened as *OMRS-LUI*) [29] is the user interface package of *OpenMRS* [30], a widely used, open-source medical record system that manages highly sensitive medical data.

Pisa-Suite contains 12 constraints generated from sanitizers detected by PISA [125]; these constraints have been used in the experimental evaluation reported in [152].

AppScan-Suite contains 8 constraints derived from the security warnings emitted by IBM Security AppScan [54], a commercial vulnerability scanner tool, when executing on a set of popular websites. The generated warnings contain traces of program statements that reflect potentially vulnerable information flows. Also these constraints have been used in the experimental evaluation reported in [152].

Kausler-Suite contains 120 constraints obtained from eight Java programs via dynamic symbolic execution. A superset of this benchmark (with 175 constraints) has been used for evaluating four string constraint solvers in the context of symbolic execution [62]; the subset used in this work contains the constraints that were successfully translated into the SMT-LIB format by the tool provided in the replication package of [62].

Cashew-Suite contains 394 distinct constraints obtained through the normalization of the constraints of the SMC/*Kaluza* benchmark by means of the *Cashew* tool [23]. The SMC/*Kaluza* benchmark [79] contains the 18896 satisfiable [5] constraints of the *Kaluza* benchmark, converted to the input format of the SMC solver [79]; the *Kaluza* benchmark contains constraints corresponding to path conditions generated from a set of JavaScript programs by a symbolic execution engine [112]. Although the *Kaluza* benchmark has been widely used for evaluating string constraint solvers in the past, its high degree of redundancy (high number of constraints equivalent in terms of satisfiability, as highlighted by the recent work on constraint normalization [23])

[5] Notice that the SMC paper [79] reports a total of 18901 satisfiable constraints, but the evaluation artifacts include only 18896 constraints.

led us to rely on *Cashew-Suite* instead. This was meant to prevent biasing the overall results with an extremely large and redundant benchmark, without any loss of information.

Since *JOACO-Suite* is the only benchmark containing the source code of Java Web applications, it was the only one used for answering *RQ1*. All five benchmarks were however used for answering *RQ2* and *RQ3*. Notice that *JOACO-Suite*, *Pisa-Suite*, and *AppScan-Suite* contain constraints derived from potentially vulnerable Web applications, whereas *Kausler-Suite* and *Cashew-Suite* have been used to evaluate string constraint solvers from a general standpoint (not necessarily related to security analysis).

We established the ground truth (i.e., whether a path is vulnerable or not) of the constraints in *JOACO-Suite* in the following way. *WebGoat* and *Bodgeit* are deliberately in-secured applications for teaching security vulnerabilities and, hence, they already provided the ground truth. However, as explained in Section 5.5.2.1, *JOACO* was able to detect four previously unknown vulnerabilities in *Bodgeit*. Since *TPC-APP*, *TPC-C* and *TPC-W* are standard benchmarks for testing vulnerability detection tools for Web services, their ground truth was available. Although *Pebble* and *Roller* have been already used as benchmarks for vulnerability detection, no ground truth was available; therefore, we consulted the US NVD [91] and confirmed the reported vulnerabilities, by exploiting them in the deployed applications and by locating their corresponding paths in the source code. *RAP*, *PSH*, *Regain* and *OMRS-LUI* did not have any recent entries in NVD; therefore, we established the ground truth by manually inspecting their source code and verified potential vulnerabilities by exploiting them in the deployed applications.

The ground truth of *Pisa-Suite* and *AppScan-Suite* was established in [152]. As for *Kausler-Suite*, its constraints are all satisfiable since they were generated by dynamic symbolic execution. The ground truth of *Cashew-Suite* was established by running *Cashew* [23], which counts the number of models for every constraint: a model count greater zero indicates satisfiability.

We ran the experiments on a machine equipped with an Intel i7 $2.4\,\mathrm{GHz}$ processor, $8\,\mathrm{GB}$ memory, running Apple Mac OS X 10.13.

The applications and the constraints included in the benchmark used for the evaluation are available on the tool web site [128].

5.5.2 Experimental Results

5.5.2.1 Effectiveness of Vulnerability Detection

To answer *RQ1*, we executed *JOACO* on the *JOACO-Suite* benchmark and compared it with two state-of-the-art vulnerability detection tools for Java Web applications: *LAPSE+* [98] and *SFlow* [53][6]. Similarly to *JOACO*, *LAPSE+* and *SFlow* provide an end-to-end solution to detect vulnerabilities, since they take as input the source code of an application and produce a vulnerability report. Both tools are based on taint analysis and thus require to specify sources, sinks, and sanitization procedures. More specifically, *LAPSE+* requires users to specify (similarly to *JOACO*) the bytecode signatures of sources, sinks, and sanitization functions in a library file; *SFlow* requires users to manually annotate the functions in code corresponding to sources and sinks[7]. Hence, before executing these tools on the *JOACO-Suite* benchmark, we specified/anno-

[6] *SFlow* has been shown [53] to perform better than Andromeda [137], a commercial product from IBM.

[7] In the case of *SFlow*, we could not annotate sanitization operations because, different from what is reported in the corresponding paper [53], the implementation of *SFlow* does not support "untaint" annotations for sanitization functions.

Table 5.6: Vulnerable and non-vulnerable paths in the applications contained in the *JOACO-Suite* benchmark. A vulnerable path corresponds to a single vulnerability.

Application	LOC	# Paths	Vulnerable					Non-Vulnerable				
			XML	XPATH	XSS	LDAP	SQL	XML	XPATH	XSS	LDAP	SQL
WebGoat	24,608	15	1	2	0	0	8	0	1	0	0	3
Roller	52,433	13	0	0	3	0	0	7	0	3	0	0
Pebble	36,592	13	2	0	4	0	0	1	0	6	0	0
Regain	23,182	6	0	0	3	0	0	0	0	3	0	0
PSH	1,964	4	1	0	0	0	0	1	0	2	0	0
TPC-APP	2,082	12	0	0	0	0	6	0	0	0	0	6
TPC-C	9,184	34	0	0	0	0	30	0	0	0	0	4
TPC-W	2,470	6	0	0	0	0	3	0	0	0	0	3
RAP	442	1	0	0	0	1	0	0	0	0	0	0
Bodgeit	3,376	21	0	0	9	0	9	0	0	2	0	1
OMRS-LUI	34,074	4	0	0	4	0	0	0	0	0	0	0
			4	2	23	1	56	9	1	16	0	17
Total	190,407	129	86					43				

Table 5.7: Comparison of the effectiveness in vulnerability detection on the *JOACO-Suite* benchmark among *LAPSE+*, *SFlow*, and *JOACO* (*tp*: true positives, *tn*: true negatives, *fp*: false positives, *fn*: false negatives, *pd*: recall, *pr*: precision, ↯: failing cases, ⊘: timeout cases, $t(\mathrm{s})$: execution time with constraint preprocessing switched on (+*Opt*) and switched off (−*Opt*).

App	*LAPSE+*								*SFlow*								*JoanAudit+CVC4+ACO-Solver*									*JOACO*									$t(\mathrm{s})$	
	tp	*tn*	*fp*	*fn*	*pd*	*pr*	↯	$t(\mathrm{s})$	*tp*	*tn*	*fp*	*fn*	*pd*	*pr*	↯	$t(\mathrm{s})$	*tp*	*tn*	*fp*	*fn*	*pd*	*pr*	↯	⊘	$t(\mathrm{s})$	*tp*	*tn*	*fp*	*fn*	*pd*	*pr*	↯	⊘	−*Opt*	+*Opt*
WebGoat	5	3	1	6	45	83	6	7.4	9	4	0	2	82	100	0	26.4	11	4	0	0	100	100	0	0	224.0	11	4	0	0	100	100	0	0	389.9	362.2
Roller	0	10	0	3	0	–	7	4.9	1	3	7	2	33	13	0	8.3	3	10	0	0	100	100	0	10	1228.1	3	10	0	0	100	100	0	0	244.0	210.7
Pebble	1	6	1	5	17	50	8	13.9	6	1	6	0	100	50	0	4.8	6	7	0	0	100	100	0	5	660.1	6	7	0	0	100	100	0	5	708.1	689.6
Regain	0	3	0	3	0	–	6	0.8	0	3	0	3	0	–	0	7.5	3	3	0	0	100	100	0	0	14.6	3	3	0	0	100	100	0	0	87.5	74.3
PSH	0	3	0	1	0	–	4	0.4	0	3	0	1	0	–	0	2.5	1	3	0	0	100	100	0	2	264.4	1	3	0	0	100	100	0	2	288.7	286.4
TPC-APP	2	6	0	4	33	100	7	4.2	5	5	1	1	83	83	0	7.7	5	6	0	1	83	100	0	3	327.0	5	6	0	1	83	100	0	0	171.7	151.9
TPC-C	0	4	0	30	0	–	0	8.8	0	4	0	30	0	–	0	7.9	30	4	0	0	100	100	0	1	152.5	30	4	0	0	100	100	0	0	661.9	566.4
TPC-W	0	3	0	3	0	–	1	2.8	3	3	0	0	100	100	0	4.8	3	3	0	0	100	100	0	0	12.7	3	3	0	0	100	100	0	0	27.3	71.9
RAP	0	0	0	1	0	–	1	0.2	0	0	0	1	0	–	0	1.3	1	0	0	0	100	100	0	0	37.7	1	0	0	0	100	100	0	0	36.8	51.2
Bodgeit	14	0	3	4	78	82	0	14.8	0	3	0	18	0	–	21	7.4	14	3	0	4	78	100	0	3	446.6	17	3	0	1	94	100	0	0	337.3	297.5
OMRS-LUI	0	0	0	4	0	–	0	4.8	0	0	0	4	0	–	4	1.4	4	0	0	0	100	100	0	0	23.3	4	0	0	0	100	100	0	0	42.1	36.6
Total	22	38	5	64	26	81	40	62.9	24	29	14	62	28	63	25	80.0	81	43	0	5	94	100	0	24	3391.0	84	43	0	2	98	100	0	7	2995.4	2798.7

tated all the functions of the applications in the benchmark so that all the tools were configured to consider the same source/sink and sanitization signatures. The input for all tools was *the complete source code* of all the (Web) applications contained in the *JOACO-Suite* benchmark.

We also compared *JOACO* with our previous work *JoanAudit+CVC4+ACO-Solver*, i.e., the combination of our security slicing tool *JoanAudit* (see Chapter 3) with *CVC4+ACO-Solver*. *CVC4+ACO-Solver* is the combination of the *CVC4* solver with *ACO-Solver*, which was shown to be the best performing constraint solver (in the context of vulnerability detection) when compared

with *CVC4*, *Z3-str2*, and *Z3-str2+ACO-Solver* (see Chapter 4). *ACO-Solver* can be seen as the predecessor of *JOACO-CS*: it does not include any constraint preprocessing and its automata-based solving module supports a limited set of string operations; furthermore, unlike *JOACO-CS*, *ACO-Solver* relies on an external constraint solver. We set the time-out for *ACO-Solver* in *CVC4+ACO-Solver* and for *JOACO-CS* in *JOACO* to $120\,\mathrm{s}$.

Table 5.7 shows the evaluation results. Columns tp, tn, fp, and fn denote, respectively, true positives (number of vulnerable cases correctly identified), true negatives (number of non-vulnerable cases correctly identified), false positives (number of non-vulnerable cases reported as vulnerable), false negatives (number of vulnerable cases not detected). Column pd reports the *recall*, i.e., the percentage of vulnerable cases detected among the total vulnerable cases, and is computed as $pd = tp/(tp + fn) * 100$. Column pr reports the *precision*, i.e., the percentage of correctly identified vulnerable cases among the total, reported vulnerable cases, and is computed as $pr = tp/(tp+fp)*100$. Column $\lightning$ indicates the number of failing cases, i.e., the number of paths for which the analysis resulted in a run-time error, and column $\oslash$ indicates the number of time-out cases, i.e., the number of cases that took longer than $120\,\mathrm{s}$ to analyze.

Notice that, in the context of vulnerability detection, when there is a failing or time-out case, the tool neither detects a vulnerability nor produces a false alarm. Hence, failing and time-out cases may result either in a false negative or in a true negative, depending on whether the path is actually vulnerable.

We answer *RQ1* by examining the recall, the precision, and the number of failing cases in Table 5.7. *LAPSE+* detected 22 vulnerabilities (true positives) and did not detect (i.e., produced false negatives for) 64 vulnerabilities; *SFlow* detected 24 vulnerabilities and did not detect 62 vulnerabilities. These values translate into a recall value of 26% for *LAPSE+* and 28% for *SFlow*. Upon manual inspection we noticed that the false-negative cases were mainly due to improper input propagation across functions. *SFlow* also failed to analyze all web programs included in *Bodgeit* and *OMRS-LUI* because of compilation errors. *JoanAudit+CVC4+ACO-Solver* detected 81 vulnerabilities and missed 5 vulnerable cases, achieving a recall of 94%. *JOACO* achieved a high recall of 98%, detecting 84 vulnerabilities and missing only 2 vulnerable cases in *TPC-APP* and *Bodgeit*. *JOACO* missed one vulnerability for *TPC-APP* because it was unable to generate the attack condition due to the presence of constraints on Java collections, which are not supported in the current version of *JOACO*. For *Bodgeit*, *JOACO* did not report an SQL injection because the threat model 11 of Table 2.1 does not consider as vulnerable a user input sanitized through the `parseInt` method.

In terms of precision, *LAPSE+* reported 5 false positives and failed to analyze 40 cases, resulting in a precision of 81%; *SFlow* reported 14 false positives and failed in 25 cases, resulting in a precision of 63%. In both cases, the false positives were mainly due to the lack of constraint solving capabilities in these tools. *JoanAudit+CVC4+ACO-Solver* achieved a precision of 100%, with no false positives and failing cases; however, it timed-out in 24 cases, out of which 21 were non-vulnerable and 3 were vulnerable.

JOACO achieved 100% precision, with no failing cases and only 7 time-out cases, which are all UNSAT cases and hence could not be solved by the search-based solver. Compared to *JoanAudit+CVC4+ACO-Solver*, *JOACO* could handle 17 more cases without running into time-outs.

We remark that *JOACO-Suite* is an extended version of the benchmark used in Section 4.5. For the two new applications added to the benchmark (*Bodgeit* and *OMRS-LUI*), *JOACO* was

Table 5.8: Comparison of the effectiveness in constraint solving among *JOACO-CS* (with constraint preprocessing switched on and off), *Z3-str3*, *CVC4* and *CVC4+ACO-Solver* (✓: correct answers, X: incorrect answers, **?**: unknown cases, ↯: failing cases, ⊘: time-outs, S: search-based solver invocation).

Suite	*Z3-str3*									*CVC4*									*CVC4+ACO-Solver*										*JOACO-CS*								
	SAT		UNSAT		TOTAL		?	↯	⊘	SAT		UNSAT		TOTAL		?	↯	⊘	SAT		UNSAT		TOTAL		?	↯	⊘	S	SAT		UNSAT		TOTAL		?	↯	⊘	S
	X	✓	X	✓	X	✓				X	✓	X	✓	X	✓				X	✓	X	✓	X	✓					X	✓	X	✓	X	✓				
JOACO	0	14	0	20	0	34	52	35	8	1	72	1	20	2	92	0	35	0	1	81	1	22	2	103	0	0	24	33	0	85	0	37	0	122	0	0	7	7
Pisa	0	8	0	4	0	12	0	0	0	0	6	0	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	8	0	4	0	12	0	0	0	0
AppScan	0	8	0	0	0	8	0	0	0	0	3	0	0	0	3	0	0	5	0	3	0	0	0	3	0	0	5	0	0	8	0	0	0	8	0	0	0	0
Kausler	0	118	0	0	0	118	1	1	0	0	117	0	0	0	117	0	0	3	0	117	0	0	0	117	0	0	3	0	0	120	0	0	0	120	0	0	0	0
Cashew	0	381	0	12	0	393	0	1	0	0	381	0	12	0	393	1	0	0	0	382	0	12	0	394	0	0	0	0	0	382	0	12	0	394	0	0	0	0
Total	0	529	0	36	0	565	53	37	8	1	579	1	32	2	611	1	35	14	1	589	1	34	2	623	0	0	38	33	0	603	0	53	0	656	0	0	7	7

able to detect 4 previously unknown vulnerabilities (1 XSS and 3 SQLi vulnerabilities) for *Bodgeit* and 4 previously unknown XSS vulnerabilities for *OMRS-LUI*. *LAPSE+* detected only 3 of the new vulnerabilities for *Bodgeit* and could not detect any of the new vulnerabilities for *OMRS-LUI*; as mentioned above, *SFlow* could not analyze any program included in *Bodgeit* and *OMRS-LUI*. The new vulnerabilities found in *Bodgeit* have been reported on the project web site[8]; the new vulnerabilities for *OMRS-LUI* have been reported and confirmed by the *OpenMRS* developers.

We also compared the four tools in terms of execution time; the detailed results are shown in columns $t(\mathrm{s})$ of Table 5.7. *LAPSE+* took $62.9\,\mathrm{s}$; *SFlow* took $80.0\,\mathrm{s}$; *JoanAudit+CVC4+ACO-Solver* took $3391.0\,\mathrm{s}$; *JOACO*, with constraint preprocessing enabled (+*Opt*), took $2798.7\,\mathrm{s}$. The execution time of *JOACO* is much larger than that of *LAPSE+* and *SFlow*, since it includes several steps such as security slicing and constraint solving; nevertheless, such a large time is not practically relevant for the purpose of vulnerability detection since such analysis is performed when new code is committed and is not required to provide immediate feedback. Furthermore, *JOACO* is about 17.5% faster than *JoanAudit+CVC4+ACO-Solver*.

The answer to *RQ1* is that the proposed approach implemented in *JOACO* is highly effective (achieving 98% recall, 100% precision) in detecting injection vulnerabilities; it performs much better than state-of-the-art vulnerability detection tools, yielding a higher recall (+70%–72%) and precision (+19%–37%), with no failing cases. This high effectiveness in vulnerability detection comes at the cost of a higher execution time, which is however practically acceptable. Compared with our previous work *JoanAudit+CVC4+ACO-Solver*, *JOACO* detected more vulnerabilities, had much less time-out cases, and was faster.

5.5.2.2 Effectiveness of String Constraint Solving

To answer *RQ2*, we compared the constraint solving capabilities of *JOACO* when used in the stand-alone solver mode (dubbed *JOACO-CS*) with three state-of-the-art constraint solvers:

[8] Issues #17–#20 on `https://github.com/psiinon/bodgeit/`.

CVC4 (version 1.4) [74], *Z3-str3* (version 4.6.0) [21], and our previous work *CVC4+ACO-Solver* presented in Chapter 4.

For the comparison, we used the constraints contained in all five benchmark suites, for a total of 663 constraints. We set the time-out for solving each constraint to $600\,\mathrm{s}$.

Table 5.8 shows the evaluation results. For each solver, we indicate the number of correct (column ✓) and incorrect (column X) answers returned by the tool, grouped by the cases "SAT" and "UNSAT", as well as the total; column **?** indicates the number of cases for which the solver returned "UNKNOWN"; column ↯ indicates the number of cases for which the solver execution failed, due to a run-time error or crash; column ⊘ indicates the number of cases in which the solver timed out; column S indicates the number of constraints for which the search-based solver had to be invoked.

We answer *RQ2* by examining the number of correct and incorrect results, and the number of unknown/failing/time-out cases in Table 5.8.

For *JOACO-Suite*, *JOACO-CS* was the most effective solver, with 122 correct results (out of 129) and no unknown/failing cases. The 7 time-out cases are the ones discussed above for the same benchmark in the answer to *RQ1*: they are UNSAT cases and therefore, the search-based solver could not find satisfying solutions for them. By contrast, *Z3-str3* yielded 34 correct with 52 unknown cases, 35 failing cases, and 8 time-outs; *CVC4* yielded 92 correct results and 2 incorrect ones, with 35 failing cases; *CVC4+ACO-Solver* yielded 103 correct results and 2 incorrect ones, with 24 time-out cases. The failing cases of *CVC4* and *Z3-str3* were due to unsupported operations.
For *Pisa-Suite*, *Z3-str3* and *JOACO-CS* were the most effective solvers, solving all constraints correctly; by contrast, both *CVC4* and *CVC4+ACO-Solver* had six timeouts.
AppScan-Suite could be correctly solved by both *JOACO-CS* and *Z3-str3*, whereas both *CVC4* and *CVC4+ACO-Solver* had 5 timeouts.
For *Kausler-Suite*, *JOACO-CS* was the most effective solver as well, solving all the 120 constraints correctly. On the other hand, *CVC4* and *CVC4+ACO-Solver* yielded 117 correct results and 3 time-out cases; *Z3-str3* yielded 118 correct results, one unknown and one failing case.
For *Cashew-Suite*, *CVC4+ACO-Solver* and *JOACO-CS* solved all the constraints correctly. *CVC4* reported one unknown case; *Z3-str3* had one failing case.

To sum up, even if we disregard our own benchmark *JOACO-Suite* (which is the only one with constraints with unsupported operations) and consider only the other four benchmarks (which contain only supported operations), *CVC4* correctly solved 519 constraints and had 14 time-outs (and also one unknown case); *Z3-str3* correctly solved 531 constraints and had one unknown and two failing cases. Instead, *JOACO-CS* correctly solved 534 constraints with no time-outs. Also, our previous work *CVC4+ACO-Solver* solved *less* constraints (and had more time-outs) than *JOACO-CS*.

We also compared the constraint solving time of the four tools; the results are shown in Table 5.9, together with the number of constraints in each benchmark. In total, *Z3-str3* took $5961\,\mathrm{s}$ ($\approx 1.5\,\mathrm{h}$) and correctly solved 565 cases; *CVC4* took $9125\,\mathrm{s}$ ($\approx 2.5\,\mathrm{h}$) and correctly solved 611 cases; *CVC4+ACO-Solver* took $30\,797\,\mathrm{s}$ ($\approx 8.5\,\mathrm{h}$) and correctly solved 623 cases; *JOACO-CS* took $16\,716\,\mathrm{s}$ ($\approx 4.5\,\mathrm{h}$) with constraint preprocessing enabled (+*Opt*) and correctly solved 656 cases. The average execution time for solving one constraint (computed as $\frac{\mathit{Total\ time}}{\#\mathit{constraints}}$) is $9.0\,\mathrm{s}$ for *Z3-str3*, $13.8\,\mathrm{s}$ for *CVC4*, $46.5\,\mathrm{s}$ for *CVC4+ACO-Solver*, $25.2\,\mathrm{s}$ for *JOACO-CS* (+*Opt*). On average, our approach is $1.8\times$ slower than the most effective state-of-the-art solver (*CVC4*), and about $2.8\times$ slower

Table 5.9: Execution time (in seconds) for *Z3-str3*, *CVC4*, *CVC4+ACO-Solver* and *JOACO-CS* with constraint preprocessing switched off (−*Opt*) and constraint preprocessing switched on (+*Opt*)

Suite	#Constraints	*Z3-str3*	*CVC4*	*CVC4+ACO-Solver*	*JOACO-CS* (−*Opt*)	*JOACO-CS* (+*Opt*)
JOACO	129	4949.0	69.1	14505.4	5999.4	5707.9
Pisa	12	4.5	3964.8	4006.8	212.9	178.3
AppScan	8	10.5	3002.2	3026.4	168.2	144.0
Kausler	120	945.8	2043.5	8804.3	6079.7	5031.3
Cashew	394	51.4	45.6	454.5	6827.9	5654.9
Total	663	5961.1	9125.1	30797.4	19288.2	16716.4
Avg. Time		9.0	13.8	46.5	29.1	25.2

than *Z3-str3*. *JOACO-CS* is about $1.8\times$ faster than our previous work *CVC4+ACO-Solver*; this is due to the larger number of string operations supported by *JOACO-CS*, which reduces the number of constraints (from 33 to 7) for which it is necessary to invoke the search-based solver. Nevertheless, *JOACO-CS* could solve the highest number of constraints in our benchmarks.

The answer to RQ2 is that the proposed constraint solving approach implemented by *JOACO-CS* is highly effective in string constraint solving and performs similarly to or better (+7%–14% more correctly solved cases) than state-of-the-art string constraint solvers, including our previous work, depending on the benchmark considered. In terms of execution time, *JOACO-CS* is $1.8\times$ slower than the fastest, state-of-the-art constraint solver. However, since *JOACO-CS* can solve more cases and constraint solving is typically an offline activity, with no stringent time requirements, we consider this slowdown as practically acceptable.

5.5.2.3 The Role of Constraint Preprocessing

To answer *RQ3*, we re-ran all the experiments conducted for answering *RQ1* and *RQ2* by using *JOACO* and *JOACO-CS* with preprocessing disabled (denoted by −*Opt*); we then compared the resulting values for the execution time with the ones obtained with the preprocessing enabled (denoted by +*Opt*).

For the use case of vulnerability detection, columns −*Opt* and +*Opt* in Table 5.7 show that enabling the constraint preprocessing led to reduction of about $200\,\mathrm{s}$ in execution time (from $2995.4\,\mathrm{s}$ down to $2798.7\,\mathrm{s}$) corresponding to a relative reduction of about 7%. For the use case of string constraint solving, columns *JOACO-CS* (−*Opt*) and *JOACO-CS* (+*Opt*) in Table 5.9 show that *JOACO-CS* with constraint preprocessing disabled took $19\,288.2\,\mathrm{s}$, whereas it took only $16\,716.4\,\mathrm{s}$ with constraint preprocessing enabled, corresponding to an execution time reduction of about 15%.

Since constraint preprocessing only impacts the efficiency of constraint solving, it has a higher impact on the execution time of *JOACO-CS* for string constraint solving (see Table 5.9) than for the case of vulnerability detection with *JOACO*, which also includes the security slicing step (see Table 5.7).

The answer to *RQ3* is that constraint preprocessing has a positive impact on the execution time of our approach, with reductions ranging between 7% and 15% depending on the use case.

5.6 Summary

In this chapter, we presented an integrated analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving.

This work addresses the challenge of analyzing the source code of a Java Web application for detecting injection vulnerabilities in a scalable and effective way. We have proposed an integrated approach that seamlessly combines static analysis-based security slicing with hybrid constraint solving, that is constraint solving based on a combination of automata-based solving and meta-heuristic search (Ant Colony Optimization). We use static analysis to extract minimal program slices from Web programs relevant to security and to generate the attack conditions, i.e., conditions necessary for the slices to be vulnerable. We then apply a hybrid constraint solving procedure to determine the satisfiability of attack conditions and thus detect vulnerabilities.

The experimental results, using a benchmark comprising 11 diverse and representative Web applications, show that our approach (implemented in the *JOACO* tool) is significantly more effective at detecting injection vulnerabilities than state-of-the-art approaches, achieving 98% recall and does so without producing any false alarm. We also compared the constraint solving module of our approach with state-of-the-art constraint solvers, using five different benchmarks; our approach correctly solved the highest number of constraints (656 out of 663), without producing any incorrect result, and was the one with the least time-outs and failing cases. In both scenarios, the execution time was practically acceptable, given the offline nature of vulnerability detection.

Chapter 6

Related Work

This chapter explains the related work and is organized as follows: Section 6.1 highlights the related work for security slicing (presented in Chapter 3); Section 6.2 illustrates the related work to search-driven constraint solving (presented in Chapter 4) and the integrated approach for vulnerability detection (presented in Chapter 5).

6.1 Security Slicing and Auditing

Related work that deal with the security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities can be broadly categorized into two areas: *static* taint analysis and program slicing approaches.

6.1.1 Static Taint analysis

Static taint analysis approaches label data from input sources as tainted data and then detect vulnerabilities if the tainted data flows into sinks — which may be exploited by tainted data — without passing through any sanitization function (declassifier). Implementations of static taint analysis are available for Java Web systems [3, 78, 98, 138, 137, 53], for PHP Web systems [60, 143, 141, 92, 83], and for Android systems [13].

In general, there are three key differences between static taint analysis approaches and our security slicing approach. First, static taint analysis approaches tend to focus on data-flow based tainting only, and do not consider control-dependency information. This information is often essential for correctly identifying vulnerabilities or auditing the correctness of input sanitization procedures, since selection statements are often used to check user inputs. For example, consider the code snippet below, corresponding to a sampled, simplified slice, extracted from *WebGoat*:

```
1 String employeeId = req.getParameter('id'); // SOURCE
2 if(Integer.parseInt(employeeId) == EMPLOYEE_ID))
3  results = stmt.executeQuery("SELECT␣*␣FROM␣employee␣WHERE␣userid␣=" + employeeId); // SINK
```

In the above example, a taint analysis approach would falsely report a vulnerability. More specifically, it would detect a data-flow from the input source at line 1 to the sink at line 3, without considering that the sanitization achieved through the call to `parseInt` at line 2 would have an impact on the value of `employeeId` itself. By contrast, our approach correctly identifies the path from line 1 to line 3 as secure due to the presence of the `parseInt` declassifier; hence, it does not report a vulnerability. In general, lack of support for control-flow dependencies can be the source of many false positive results: Jovanovic et al.'s taint analysis tool [60] reported five false positives; Tripp et al. [137] reported 40% false positives on analyzing *WebGoat*; Shar and Tan [118] also reported that Livshits and Lam's taint analysis approach [78] yielded 20% false positives due to missing control-dependency information. Although there are some taint analysis approaches [28, 65, 114, 61, 154] that analyze control-dependency information, but they support programming languages different from Java and/or do not address injection vulnerabilities (with the exception of Dytan [28], which addresses SQLi in the context of dynamic taint analysis for x86 code).

Second, declassification is the only form of filtering provided by taint analysis approaches (e.g., as in [92]) whereas our approach additionally filters irrelevant and known-good library functions and also fixes some of the vulnerabilities automatically.

Last, our approach specifically targets XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. Current taint analysis-based approaches address only SQLi and/or XSS. To the best of our knowledge, only Pérez et al. [98] readily address XMLi, XPathi, and LDAPi for Java Web systems. However, since Pérez et al.'s work is not evaluated, it is difficult to verify its effectiveness. Medeiros et al. [83] readily address XPathi and LDAPi but for PHP Web systems. It is possible to adapt existing approaches to support XMLi, XPathi, and LDAPi and even equip them with our proposed filtering mechanisms. However, since developers are often not security experts, these tasks may not be trivial. By contrast, our tool is already configured with an extensive library of input sources, sinks, and declassifiers specific to these vulnerabilities and thus, it can be used out-of-the-box.

6.1.2 Program slicing

Krinke [67] proposes barrier slicing approaches that could allow auditors to filter specific parts of the program that are known to be correct. Our approach makes use of this idea to prune Java libraries that are irrelevant to our security auditing purposes.

Despite the various slicing approaches proposed in the literature, in practice there are only two slicers that can handle all Java features: *Indus* [59] and *Joana* [47]. *Indus* is built on top of *Soot* [139], a Java bytecode analysis framework, and is less precise than *Joana*, since it does not fully support interprocedural slicing [47]. As discussed in Section 3.1, *Joana* provides a sound and precise approach for computing slices and chops. As our approach and tool are built on top of *Joana*, we have the same advantages. However, *Joana* only generates slices for generic tasks like checking information flow and debugging. By contrast, we provide additional techniques for pruning statements in the slices produced by *Joana* and target security auditing of vulnerabilities. Therefore, *Joana* represents our baseline for comparison.

Shar and Tan [118] propose a program slicing-based approach for auditing the implemented defense features to prevent XSS. The approach of Yamaguchi et al. [144, 145] extracts abstract syntax trees and program dependence graphs relevant to auditing buffer overflow vulnerabili-

ties in C/C++ code. The key difference between these approaches and ours is that they do not focus on minimizing the size of the extracted code, because their main objective is to extract all the possible defense features. By contrast, we extract all the features relevant for security auditing and yet, we also minimize the size of the extracted code by filtering irrelevant or secure code, making security auditing scalable and practical.

Backes et al. [15] present a program slicing-based approach for auditing privacy data leakage issues in Android code. Similarly to our approach, they also reduce SDG size by filtering known-good and irrelevant library code. But unlike our approach, they do not consider declassification and automated fixing. Further, as our objectives are different, the specifications of sources, sinks, and library APIs are also different. Hassanshahi et al. [49] propose an approach for detecting *Web-to-App* Injection (W2AI) attacks, an attack type where an adversary can exploit a vulnerable app through the bridge that enables interaction between the browser and apps installed on Android phones. Like our approach, they also make use of program slicing based on the ICFG in conjunction with a pre-defined set of sources and sinks. However, the main objective of their work is the detection of 0-day W2AI vulnerabilities rather than helping security analysts to audit source code for finding and fixing vulnerabilites of various kind.

6.2 Hybrid Analysis Framwork

Our proposed approach is related to work done in the areas of code-based security analysis, penetration testing, (string) constraint solving, constraint solving through heuristic search, and search-based test input generation for string data types.

Code-based security analysis. This category includes two types of approaches: taint analysis and symbolic execution. Approaches bases on taint analysis (such as [78, 60, 46, 138, 137, 53]) check whether application inputs are used in sinks without passing through known sanitization functions. However, these approaches tend to generate many false alarms [17, 6] since they cannot reason about the implementation of sanitization functions. Some approaches [140, 141] incorporate string analysis into taint analysis, improving the precision in the analysis of SQLi and XSS vulnerabilities. Other approaches [17, 150, 148] reason about the adequacy of input sanitization code by combining taint analysis and string constraint solving using finite state automata operations.

Approaches based on symbolic execution [64, 112, 153] perform (dynamic) symbolic execution on programs and generate path conditions. They then use a constraint solver to check these conditions and determine whether inputs used in sinks may contain security attack values. These approaches, which rely on (string) constraint solving, exhibit the same limitations (e.g., limited support for complex string operations) of constraint solvers, which are discussed further below. In addition, these approaches switch to *dynamic* symbolic execution for scalability when encountering the path explosion problem; however, such a strategy may lead to omit to analyze certain parts of the program and thus, miss vulnerabilities. By contrast, our approach applies security slicing to extract only program parts relevant to security; this greatly improves scalability without sacrificing vulnerability detection effectiveness.

Penetration testing. Penetration testing tools like Acunetix [2], BurpSuite [99], and AppScan [54] are useful for detecting the presence of vulnerabilities in Web programs. Antunes and Vieira [6] evaluated these tools and observed that penetration testing approaches miss vulnerabilities and are in generally less accurate than taint analysis approaches. More specifically,

penetration testing cannot detect vulnerabilities that require to craft the corresponding attack values in order to exploit the weaknesses of input sanitization functions. On the other hand, our approach, being based on constraint solving, returns a concrete attack value for the input (i.e., a solution) when it identifies a vulnerability.

(String) constraint solving. There are many constraint solvers that provide, to a certain degree, support for strings: bit-vector-based solvers like Hampi [63], Kaluza [112], and Utopia [11]; automata-based solvers like Violist [72], Stranger [147, 149], ABC [14], StrSolve [51], Pass [73], StringGraph [103], and JST [41]; word-based solvers like Norn [1], S3 [136], and the aforementioned Sushi, *CVC4*, *Z3-str2* and *Z3-str3*. Among them, Stranger, JST, StringGraph, S3, *CVC4*, *Z3-str2* and *Z3-str3* support the largest number of string operations (e.g., `startsWith`, `endsWith`, `replace`, `replaceAll`, `length`, and `matches`) that are essential in the context of vulnerability detection; they also support numeric constraints. Although Hampi and Kaluza have been widely used as benchmarks for evaluating other solvers (see [74, 152, 136, 1]), they actually support only a smaller set of string operations than the solvers listed above; also, Hampi does not support numeric constraints. Support for regular expressions (which are usually used in attack specifications) is only provided — often in a limited form — by Sushi, Stranger, ABC, Kaluza, S3, *Z3-str2*, *Z3-str3* and *CVC4*. Nevertheless, none of them provides full support for a complete string function library of a modern programming language or for sanitization libraries like OWASP ESAPI and Apache Commons Lang. This means that they fail when they encounter an unsupported operation in an input constraint; in turn, this may lead to missing vulnerabilities. By contrast, in our approach we use a search-based meta-heuristic algorithm to handle unsupported operations.

Constraint solving through heuristic search. Heuristic search has been already proposed [34] for solving non-linear arithmetic constraints with operations from unsupported numeric libraries; the heuristics is optimized to explore an n-dimensional space over real numbers. In our approach we focus on solving constraints with string/mixed and integer operations; the search heuristics is optimized, in terms of search strategy and fitness functions, for these kinds of constraints. Further, the approach in [34] is evaluated in terms of coverage of test generators, while we evaluated our approach in the context of vulnerability detection.

Search-based test input generation for string data types. There are a few proposals [4, 82, 39] that apply a search-based approach (typically genetic algorithms) for generating test cases in the form of string input values, in the context of satisfaction of branch coverage criteria. Their goal is to improve coverage by driving the search for string values, either with useful seed values [4, 82] or by hybridizing global search and local search [39]. In our case, attack conditions (which include full path conditions and attack specifications) are much more complex than branch conditions and thus we need to reduce the search space. Since we rely on automata-based solvers for search space reduction, our search algorithm works on automata and, as a result, we had to devise a search strategy that is effective on graph representations. This was the main reason to select Ant Colony Optimization, which resulted in a significantly different search strategy than the ones proposed in the above-mentioned approaches.

Automated Mitigation. There are approaches that automatically mitigate code injection problems by sanitizing potentially malicious user inputs [113, 111] or by inserting runtime mechanisms that check against security policies [22, 121]. CSAS [111] automatically inserts sanitization routines into the code generated through Web templating frameworks, which ensure untrusted user inputs are properly sanitized. ScriptGuard [113] learns which sanitizers to use for certain

program paths during a training phase, infers incorrect sanitizations, and fixes them by applying the correct sequences of sanitizers. Synode [121] statically computes templates of the values passed to the (Node.js) APIs and synthesizes a security policy from these templates, which is used to detect potential attacks at runtime. XSS-Guard [22] learns legitimate scripts a Web application may create by observing normal operation behaviors and removes any anomalous script at runtime.

Similar to these approaches, *JOACO* also applies a light-weight automatic sanitization during the security slicing step. However, by contrast, our approach does not involve dynamic analysis or runtime checks; we designed our approach based on static analyses such that it can be used by developers during software development. Furthermore, these approaches focus on XSS issues only whereas ours work on common injection issues (XSS, SQLi, XMLi, XPathi, LDAPi).

Part III

Finale

Chapter 7

Conclusions & Future Work

7.1 Conclusions

The goal of this thesis is to provide a scalable approach, based on symbolic execution and constraint solving, which aims to effectively find injection vulnerabilities in the server-side code of Java Web applications and which generates no or few false alarms, minimizes false negatives, overcomes the path explosion problem and enables the solving of complex constraints.

We have tackled the path explosion problem and the challenge of solving complex constraints by proposing an integrated hybrid vulnerability analysis framework for injection vulnerabilities in Web applications, which leverages the synergistic combination of security slicing and hybrid constraint solving.

Security slicing mitigates the path explosion problem by generating only the constraints that characterize the security slices of the program under analysis. This step allows us to identify paths and statements in the program where vulnerabilities can be exploited; this renders the remainder of the approach scalable, including symbolic execution.

The second step takes as input the attack conditions generated in the previous step, in the form of a constraint. The resulting constraint is then given as input to a hybrid constraint solver which orchestrates a constraint solving procedure for string/mixed and integer constraints with our search-based constraint solving procedure. The results yielded by the hybrid constraint solver are used to create the vulnerability report.

This remainder of this chapter is organized as follows: Section 7.2 summarizes the contributions; Section 7.3 provides directions for future work.

7.2 Contributions

In this thesis we have made the following contributions:

I *Sound and scalable security auditing.* We define a specific security slicing approach for the auditing of security vulnerabilities in the server-side code of Web applications.

II *Search-driven constraint solving.* A search-driven technique for solving string constraints with complex string operations in the context of vulnerability detection.

III *Integrated Approach.* An integrated analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving.

IV *Specialized Security Analysis.* The application of the above-mentioned techniques to detect the five most common types of injection vulnerabilities (XSS, SQLi, XMLi, XPathi, LDAPi) in the context of Java Web applications.

V *Tool support.* The implementation of the proposed techniques in prototype tools: *JoanAudit*, i.e., the tool which implements our security slicing approach, *ACO-Solver* and *JOACO-CS*, i.e., the tools that implement hybrid constraint solving, and *JOACO*, i.e., the implementation of the integrated approach for vulnerability detection.

VI *Extensive empirical evaluation.* An extensive empirical evaluation of the approaches mentioned above.

The experimental results show that our approach (implemented in the *JOACO* tool) is significantly more effective at detecting injection vulnerabilities than state-of-the-art approaches and does so without producing any false alarm.

We also compared the constraint solving module *JOACO-CS* of our approach with state-of-the-art constraint solvers, using five different benchmarks; our approach correctly solved the highest number of constraints without producing any incorrect result, and was the one with the least time-outs and failing cases. In both scenarios, the execution time was practically acceptable, given the offline nature of vulnerability detection.

7.3 Future Work

As part of future work, we will adapt our integrated vulnerability detection approach to widely-used Java Web frameworks such as *Spring* [120] or *Play Framework* [75], and we plan to integrate our approach into build management tools like *Maven* [9] or *Gradle* [35]. Furthermore, we will conduct a user study to assess the usefulness of our tool in industrial settings.

Bibliography

[1] ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., HOLÍK, L., REZINE, A., RÜMMER, P., AND STENMAN, J. Norn: An SMT solver for string constraints. In *Proceedings of CAV'15* (2015), Springer, pp. 462–469.

[2] ACUNETIX. Acunetix: Web vulnerability scanner. `https://www.acunetix.com/`, 2017.

[3] ALMORSY, M., GRUNDY, J., AND IBRAHIM, A. S. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *Proceedings of ASE* (2012), ACM, pp. 100–109.

[4] ALSHRAIDEH, M., AND BOTTACI, L. Search-based software test data generation for string data using program-specific search operators. *Softw. Test. Verif. Reliab. 16*, 3 (2006), 175–203.

[5] ANTUNES, N., AND VIEIRA, M. Soa-scanner: An integrated tool to detect vulnerabilities in service-based infrastructures. In *Proceedings of SCC* (2013), IEEE Computer Society, pp. 280–287.

[6] ANTUNES, N., AND VIEIRA, M. Assessing and comparing vulnerability detection tools for Web services: Benchmarking approach and examples. *Trans. Serv. Comput. 8*, 2 (2015), 269–283.

[7] APACHE. Apache Roller blogging application. `http://roller.apache.org/`, 2017.

[8] APACHE. StringEscapeUtils. `https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/StringEscapeUtils.html`, 2017.

[9] APACHE SOFTWARE FOUNDATION. the apache maven project. `https://maven.apache.org/`, 2018.

[10] APPELT, D., NGUYEN, C. D., BRIAND, L. C., AND ALSHAHWAN, N. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *Proceedings of ISSTA* (2014), ACM, pp. 259–269.

[11] AQUINO, A., DENARO, G., AND PEZZÈ, M. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *Proceedings of ICSE'17* (2017), IEEE, pp. 427–437.

[12] ARCURI, A. It does matter how you normalise the branch distance in search based software testing. In *Proceedings of ICST'10* (2010), IEEE, pp. 205–214.

[13] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of PLDI* (2014), ACM, pp. 259–269.

[14] AYDIN, A., BANG, L., AND BULTAN, T. Automata-based model counting for string constraints. In *Proceedings of CAV'15* (2015), Springer, pp. 255–272.

[15] BACKES, M., BUGIEL, S., DERR, E., AND HAMMER, C. Taking Android app vetting to the next level with path-sensitive value analysis. Tech. Rep. A/02/2014, Saarland University, 2014.

[16] BAILLY, G., OULASVIRTA, A., KÖTZING, T., AND HOPPE, S. Menuoptimizer: Interactive optimization of menu systems. In *Proceedings of UIST'13* (New York, NY, USA, 2013), UIST '13, ACM, pp. 331–342.

[17] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *Proceedings of S&P'08* (2008), IEEE, pp. 387–401.

[18] BENNETTS, S. The bodgeit store. `https://github.com/psiinon/bodgeit`, 2017.

[19] BENZ, F., AND KÖTZING, T. An effective heuristic for the smallest grammar problem. In *Proceedings of GECCO'13* (New York, NY, USA, 2013), ACM, pp. 487–494.

[20] BERGERETTI, J., AND CARRÉ, B. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst. 7*, 1 (1985), 37–61.

[21] BERZISH, M., ZHENG, Y., AND GANESH, V. Z3str3: A string solver with theory-aware branching. *CoRR abs/1704.07935* (2017).

[22] BISHT, P., AND VENKATAKRISHNAN, V. N. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of DIMVA'08* (Berlin, Heidelberg, 2008), Springer, pp. 23–43.

[23] BRENNAN, T., TSISKARIDZE, N., ROSNER, N., AYDIN, A., AND BULTAN, T. Constraint normalization and parameterized caching for quantitative program analysis. In *Proceedings of ESEC/FSE'17* (2017), ACM, pp. 535–546.

[24] CADAR, C., AND SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 2 (2013), 82–90.

[25] CARO, G. D. *Ant Colony Optimization and its Application to Adaptive Routing in Telecommunication Networks*. PhD thesis, Université Libre de Bruxelles, 2004.

[26] CASS, S. The 2017 Top Programming Languages. `http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`, 2017.

[27] CHRISTENSEN, A., MØLLER, A., AND SCHWARTZBACH, M. Precise analysis of string expressions. In *Proceedings of SAS'03* (2003), Springer, pp. 1–18.

[28] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of ISSTA* (2007), ACM, pp. 196–206.

[29] COMMUNITY, O. OpenMRS legacy user interface module. `https://github.com/openmrs/openmrs-module-legacyui`, 2017.

[30] COMMUNITY, O. OpenMRS project. `http://openmrs.org/`, 2017.

[31] CROES, G. A. A method for solving traveling-salesman problems. *Oper. Res. 6*, 6 (1958), 791–812.

[32] CWE. Common weakness enumeration. `http://cwe.mitre.org/`, 2017.

[33] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (1991), 451–490.

[34] DINGES, P., AND AGHA, G. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of FSE'14* (2014), ACM, pp. 425–436.

[35] DOCKTER, A. H., MURDOCH, S., FABER, P., NIEDERWIESER, D., DEBOER, L. D., AND GRÖSCHKE, R. The Gradle Build Tool. `https://gradle.org`, 2018.

[36] DORIGO, M., AND BIRATTARI, M. Ant colony optimization. In *Encycl. of Mach. Learn.* Springer, 2010, pp. 36–39.

[37] DORIGO, M., AND SOCHA, K. An introduction to ant colony optimization. Tech. Rep. TR/IRIDIA/2006-010, IRIDIA, 2006.

[38] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (1987), 319–349.

[39] FRASER, G., ARCURI, A., AND MCMINN, P. A "memetic" algorithm for whole test suite generation. *J. Syst. Softw. 103* (2015), 311–327.

[40] FU, X., POWELL, M., BANTEGUI, M., AND LI, C.-C. Simple linear string constraints. *Form. Asp. Comput. 25*, 6 (2013), 847–891.

[41] GHOSH, I., SHAFIEI, N., LI, G., AND CHIANG, W.-F. JST: An automatic test generation tool for industrial Java applications with strings. In *Proceedings of ICSE'13* (2013), IEEE, pp. 992–1001.

[42] GOOGLE. Google Code Archive. https://code.google.com/archive.

[43] GRAF, J., MOHR, M., HECKER, M., BISCHOF, S., AND BLASCHKE, T. Joana - Information Flow Control for Java. https://github.com/joana-team/joana, 2017.

[44] GREENBERG, H. J. Consistency, redundancy, and implied equalities in linear systems. *Ann. of Math. and Artif. Intell. 17*, 1 (1996), 37–83.

[45] GROUP, N. W. Pubsubhubbub core 0.4. http://pubsubhubbub.github.io/PubSubHubbub/pubsubhubbub-core-0.4.html, 2014.

[46] HALFOND, W., ORSO, A., AND MANOLIOS, P. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation. *Trans. Softw. Eng. 34*, 1 (2008), 65–81.

[47] HAMMER, C. *Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs*. PhD thesis, Karlsruhe Institute of Technology, 2009.

[48] HARMAN, M., AND MCMINN, P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Trans. Softw. Eng. 36*, 2 (2010), 226–247.

[49] HASSANSHAHI, B., JIA, Y., YAP, R. H. C., SAXENA, P., AND LIANG, Z. Web-to-application injection attacks on Android: Characterization and detection. In *Proceedings of ESORICS* (2015), Springer, pp. 577–598.

[50] HICKEY, T., JU, Q., AND VAN EMDEN, M. H. Interval arithmetic: From principles to implementation. *J. ACM 48*, 5 (2001), 1038–1068.

[51] HOOIMEIJER, P., AND WEIMER, W. StrSolve: solving string constraints lazily. *Autom. Softw. Eng. 19*, 4 (2012), 531–559.

[52] HORWITZ, S., REPS, T. W., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *Trans. Program. Lang. Syst. 12*, 1 (1990), 26–60.

[53] HUANG, W., DONG, Y., AND MILANOVA, A. Type-based taint analysis for Java Web applications. In *Proceedings of FASE'14* (2014), Springer, pp. 140–154.

[54] IBM. IBM Security Scanner: AppScan. http://www-03.ibm.com/software/products/en/appscan-source., 2017.

[55] IBM. T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net., 2017.

[56] INSTITUTE, P. 2013 cost of data breach study: Global analysis. http://www.symantec.com/content/en/us/about/media/pdfs/b-cost-of-a-data-breach-global-report-2013.en-us.pdf, 2013.

[57] JACKSON, D., AND ROLLINS, E. J. Chopping: A generalization of slicing. Tech. Rep. CMU-CS-94-169, Carnegie Mellon University, 1994.

[58] JAN, S., NGUYEN, C. D., AND BRIAND, L. C. Automated and effective testing of web services for XML injection attacks. In *Proceedings of ISSTA'16* (New York, NY, USA, 2016), ACM, pp. 12–23.

[59] JAYARAMAN, G., RANGANATH, V. P., AND HATCLIFF, J. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *Proceedings of FASE* (2005), Springer, pp. 269–272.

[60] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *Proceedings of S&P'06* (2006), IEEE, pp. 257–263.

[61] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of NDSS* (2011), The Internet Society.

[62] KAUSLER, S., AND SHERMAN, E. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of ASE'14* (2014), ACM, pp. 259–270.

[63] KIEZUN, A., GANESH, V., GUO, P. J., HOOIMEIJER, P., AND ERNST, M. D. HAMPI: A solver for string constraints. In *Proceedings of ISSTA'09* (2009), ACM, pp. 105–116.

[64] KIEZUN, A., GUO, P., JAYARAMAN, K., AND ERNST, M. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of ICSE'09* (2009), IEEE, pp. 199–209.

[65] KING, D., HICKS, B., HICKS, M., AND JAEGER, T. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of ICISS* (2008), Springer, pp. 56–70.

[66] KOREL, B. Automated software test data generation. *Trans. Softw. Eng. 16*, 8 (1990), 870–879.

[67] KRINKE, J. Slicing, chopping, and path conditions with barriers. *Springer Softw. Qual. J. 12*, 4 (2004), 339–360.

[68] L., P. D., AND K., M. A. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[69] LARANJEIRO, N., VIEIRA, M., AND MADEIRA, H. A technique for deploying robust Web services. *IEEE Trans. Serv. Comput. 7*, 1 (2014), 68–81.

[70] LECOUTRE, C. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. Wiley, 2009.

[71] LERCH, J., HERMANN, B., BODDEN, E., AND MEZINI, M. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of FSE'14* (New York, NY, USA, 2014), ACM, pp. 98–108.

[72] LI, D., LYU, Y., WAN, M., AND HALFOND, W. G. J. String analysis for Java and Android applications. In *Proceedings of ESEC/FSE'15* (2015), ACM, pp. 661–672.

[73] LI, G., AND GHOSH, I. PASS: String solving with parameterized array and interval automaton. In *Proceedings of HVC'13* (2013), Springer, pp. 15–31.

[74] LIANG, T., REYNOLDS, A., TINELLI, C., BARRETT, C., AND DETERS, M. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of CAV'14* (2014), Springer, pp. 646–662.

[75] LIGHTBEND, AND ZENGULARITY. the play framework. `https://www.playframework.com/`, 2018.

[76] LIN, S. Computer solutions of the traveling salesman problem. *Alcatel-Lucent Bell Syst. Tech. J. 44*, 10 (1965), 2245–2269.

[77] LIU, Y., AND MILANOVA, A. Practical static analysis for inference of security-related program properties. In *Proceedings of ICPC'09* (2009), IEEE, pp. 50–59.

[78] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of USENIX Security* (2005), USENIX Association.

[79] LUU, L., SHINDE, S., SAXENA, P., AND DEMSKY, B. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 565–576.

[80] MAINKA, C., JENSEN, M., IACONO, L. L., AND SCHWENK, J. Making xml signatures immune to xml signature wrapping attacks. In *Proceedings of CLOSER* (2013), Springer, pp. 151–167.

[81] MCMILLAN, K. L. An interpolating theorem prover. *Theor. Comput. Sci. 345*, 1 (2005), 101–121.

[82] MCMINN, P., SHAHBAZ, M., AND STEVENSON, M. Search-based test input generation for string data types using the results of Web queries. In *Proceedings of ICST'12* (2012), IEEE, pp. 141–150.

[83] MEDEIROS, I., NEVES, N., AND CORREIA, M. Equipping wap with weapons to detect vulnerabilities: Practical experience report. In *Proceedings of DSN* (2016), IEEE Computer Society, pp. 630–637.

[84] MERKOW, M., AND RAGHAVAN, L. *Secure and Resilient Software Development*. CRC Press, 2015.

[85] MØLLER, A., AND SCHWARZ, M. Automated detection of client-state manipulation vulnerabilities. *ACM Trans. Softw. Eng. Methodol. 23*, 4 (2014), 29:1–29:30.

[86] MOORE, R. E., KEARFOTT, R. B., AND CLOUD, M. J. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[87] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[88] MYERS, A. C., SABELFELD, A., AND ZDANCEWIC, S. Enforcing robust declassification and qualified robustness. *IOS J. Comput. Secur. 14*, 2 (2006), 157–196.

[89] NAVARRO, G. A guided tour to approximate string matching. *Comput. Surv. 33*, 1 (2001), 31–88.

[90] NAVEH, B. JGraphT. https://github.com/jgrapht/jgrapht, 2017.

[91] NIST. NIST: National vulnerability database. https://nvd.nist.gov/, 2017.

[92] NUNES, P. J. C., FONSECA, J., AND VIEIRA, M. phpsafe: A security analysis tool for oop web application plugins. In *Proceedings of DSN* (2015), IEEE Computer Society, pp. 299–306.

[93] OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. In *Proceedings of SIGSOFT/SIGPLAN PSDE* (1984), ACM, pp. 177–184.

[94] OWASP. OWASP ESAPI. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, 2017.

[95] OWASP. OWASP Top 10. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2017.

[96] OWASP. OWASP WebGoat project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2017.

[97] PEBBLE. A lightweight, open source, Java EE blogging tool. http://pebble.sourceforge.net/, 2017.

[98] PÉREZ, P. M., FILIPIAK, J., AND SIERRA, J. M. LAPSE+ static analysis security software: Vulnerabilities detection in Java EE applications. In *Proceedings of FutureTech* (2011), Springer, pp. 148–156.

[99] PORTSWIGGER LTD. Burp Suite: toolkit for Web application security testing. https://portswigger.net/burp/, 2017.

[100] PUBSUBHUBBUB. A simple, open, webhook based pubsub protocol & open source reference implementation. https://code.google.com/p/pubsubhubbub/, 2017.

[101] RAZZAQ, A., LATIF, K., AHMAD, H. F., HUR, A., ANWAR, Z., AND BLOODSWORTH, P. C. Semantic security against Web application attacks. *Inf. Sci. 254* (2014), 19–38.

[102] REDELINGHUYS, G. *Symbolic string execution*. PhD thesis, Stellenbosch: Stellenbosch University, 2012.

[103] REDELINGHUYS, G., VISSER, W., AND GELDENHUYS, J. Symbolic execution of programs with strings. In *Proceedings of SAICSIT'12* (2012), ACM, pp. 139–148.

[104] REGAIN. Regain search engine. http://regain.sourceforge.net/, 2017.

[105] REPS, T. W., AND ROSAY, G. Precise interprocedural chopping. In *Proceedings of SIGSOFT FSE* (1995), ACM, pp. 41–52.

[106] ROSA, T. M., SANTIN, A. O., AND MALUCELLI, A. Mitigating XML injection 0-day attacks through strategy-based detection systems. *IEEE Secur. & Priv. 11*, 4 (2013), 46–53.

[107] ROSSI, F., VAN BEEK, P., AND WALSH, T. *Handbook of constraint programming*. Elsevier, 2006.

[108] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J.Sel. A. Commun. 21*, 1 (2003), 5–19.

[109] SABELFELD, A., AND SANDS, D. Dimensions and principles of declassification. In *Proceedings of CSFW* (2005), IEEE Computer Society, pp. 255–269.

[110] SAFAR, K. rest-auth-proxy. `https://github.com/kamranzafar/rest-auth-proxy`, 2017.

[111] SAMUEL, M., SAXENA, P., AND SONG, D. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of CCS'11* (New York, NY, USA, 2011), ACM, pp. 587–600.

[112] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for JavaScript. In *Proceedings of S&P'10* (2010), IEEE, pp. 513–528.

[113] SAXENA, P., MOLNAR, D., AND LIVSHITS, B. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of CCS'11* (New York, NY, USA, 2011), ACM, pp. 601–614.

[114] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of SP* (2010), IEEE Computer Society, pp. 317–331.

[115] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for C. In *Proceedings of ESEC/FSE'05* (2005), ACM, pp. 263–272.

[116] SHAHRIAR, H., AND ZULKERNINE, M. Information-theoretic detection of SQL injection attacks. In *Proceedings of HASE* (2012), IEEE Computer Society, pp. 40–47.

[117] SHANNON, D., HAJRA, S., LEE, A., ZHAN, D., AND KHURSHID, S. Abstracting symbolic execution with string analysis. In *Proceedings of TAICPART-MUTATION'07* (2007), IEEE, pp. 13–22.

[118] SHAR, L. K., AND TAN, H. B. K. Auditing the XSS defence features implemented in Web application programs. *IET Softw. 6*, 4 (2012), 377–390.

[119] SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol. 10*, 2 (2001), 209–254.

[120] SPRINGSOURCE. The Spring Framework. `https://spring.io/`, 2017.

[121] STAICU, C.-A., PRADEL, M., AND LIVSHITS, B. Understanding and automatically preventing injection attacks on node.js. Tech. Rep. TUD-CS-2016-14663, Technical University Darmstadt, 2016.

[122] STÜTZLE, T., AND HOOS, H. H. Max-min ant system. *Future Gener. Comput. Syst. 16*, 9 (2000), 889–914.

[123] SU, Z., AND WASSERMANN, G. The essence of command injection attacks in Web applications. In *Proceedings of POPL* (2006), ACM, pp. 372–382.

[124] TAO, Z. Detection and service security mechanism of XML injection attacks. In *Proceedings of ICICA* (2013), Springer, pp. 67–75.

[125] TATEISHI, T., PISTOIA, M., AND TRIPP, O. Path- and index-sensitive string analysis based on monadic second-order logic. *Trans. Softw. Eng. Methodol. 22*, 4 (2013), 33:1–33:33.

[126] TEITELBAUM, T. Codesurfer. *SIGSOFT Softw. Eng. Notes 25*, 1 (2000), 99.

[127] THOMÉ, J. JoanAudit: a security slicing tool. `https://github.com/julianthome/joanaudit`, 2015.

[128] THOMÉ, J. JOACO: Vulnerability analysis through security slicing and hybrid constraint solving. `https://sites.google.com/site/joacosite/home`, 2017.

[129] THOMÉ, J., GORLA, A., AND ZELLER, A. Search-based security testing of Web applications. In *Proceedings of SBST Workshop* (2014), ACM, pp. 5–14.

[130] THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. Search-driven string constraint solving for vulnerability detection. In *Proceedings of ICSE'17* (2017), IEEE, pp. 198–208.

[131] THOMÉ, J., SHAR, L., BIANCULLI, D., AND BRIAND, L. Security slicing for auditing common injection vulnerabilities. *J. Syst. Softw. 137* (March 2018), 766–783.

[132] THOMÉ, J., SHAR, L. K., BIANCULLI, D., AND BRIAND, L. C. Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of ESEC/FSE'17* (2017), ACM, pp. 1004–1008.

[133] THOMÉ, J., SHAR, L. K., BIANCULLI, D., AND BRIAND, L. C. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. Tech. Rep. TR-SNT-2017-4, SnT Centre, 2018.

[134] THOMÉ, J., SHAR, L. K., AND BRIAND, L. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities. In *Proceedings of ISSRE'15* (2015), IEEE, pp. 553–564.

[135] THOMÉ, J. A scalable and accurate hybrid vulnerability analysis framework. In *Proceedings of ISSREW'15* (2015), IEEE, pp. 61–62.

[136] TRINH, M.-T., CHU, D.-H., AND JAFFAR, J. S3: A symbolic string solver for vulnerability detection in Web applications. In *Proceedings of CCS'14* (2014), ACM, pp. 1232–1243.

[137] TRIPP, O., PISTOIA, M., COUSOT, P., COUSOT, R., AND GUARNIERI, S. ANDROMEDA: Accurate and scalable security analysis of Web applications. In *Proceedings of FASE'13* (2013), Springer, pp. 210–225.

[138] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. TAJ: Effective taint analysis of Web applications. In *Proceedings of PLDI'09* (2009), ACM, pp. 87–97.

[139] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L. J., LAM, P., AND SUNDARESAN, V. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON* (1999), IBM, p. 13.

[140] WASSERMANN, G., AND SU, Z. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proceedings of PLDI'07* (2007), ACM, pp. 32–41.

[141] WASSERMANN, G., AND SU, Z. Static detection of cross-site scripting vulnerabilities. In *Proceedings of ICSE'08* (2008), ACM, pp. 171–180.

[142] XIE, J., CHU, B., LIPFORD, H. R., AND MELTON, J. T. ASIDE: IDE support for Web application security. In *Proceedings of ACSAC'11* (2011), ACM, pp. 267–276.

[143] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *USENIX Security* (2006), vol. 6, pp. 179–192.

[144] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of SP* (2014), IEEE Computer Society.

[145] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H., AND RIECK, K. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proceedings of CCS* (2013), ACM, pp. 499–510.

[146] YANG, G., PERSON, S., RUNGTA, N., AND KHURSHID, S. Directed incremental symbolic execution. *ACM Trans. Softw. Eng. Methodol. 24*, 1 (2014), 3:1–3:42.

[147] YU, F., ALKHALAF, M., AND BULTAN, T. Stranger: An automata-based string analysis tool for PHP. In *Proceedings of TACAS'10* (2010), Springer, pp. 154–157.

[148] YU, F., ALKHALAF, M., AND BULTAN, T. Patching vulnerabilities with sanitization synthesis. In *Proceedings of ICSE'11* (2011), ACM, pp. 251–260.

[149] YU, F., ALKHALAF, M., BULTAN, T., AND IBARRA, O. H. Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des. 44*, 1 (2014), 44–70.

[150] YU, F., SHUEH, C.-Y., LIN, C.-H., CHEN, Y.-F., WANG, B.-Y., AND BULTAN, T. Optimal sanitization synthesis for Web application vulnerability repair. In *Proceedings of ISSTA'16* (2016), ACM, pp. 189–200.

[151] ZHANG, Y., AND YAP, R. H. C. Arc consistency on n-ary monotonic and linear constraints. In *Proceedings of CP'02* (2000), Springer, pp. 470–483.

[152] ZHENG, Y., GANESH, V., SUBRAMANIAN, S., TRIPP, O., DOLBY, J., AND ZHANG, X. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Proceedings of CAV'15* (2015), Springer, pp. 235–254.

[153] ZHENG, Y., AND ZHANG, X. Path sensitive static analysis of Web applications for remote code execution vulnerability detection. In *In Proceedings of ICSE'13* (2013), IEEE, pp. 652–661.

[154] ZHU, E., LIU, F., WANG, Z., LIANG, A., ZHANG, Y., LI, X., AND LI, X. Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs. *Computers & Security 52* (2015), 51–69.